PROYECTO DE SISTEMAS INFORMÁTICOS

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

CURSO 2009/2010

MOTOR DE FÍSICA MULTIFUNCIONAL

PROFESOR DIRECTOR: Dr. Segundo Esteban San Román

AUTORES:

Jaime Sánchez-Valiente Blázquez

Santiago Riera Almagro

Matías Salinero Delgado

Resumen.

Tanto la industria de los videojuegos como la de la animación requieren motores físicos que permitan reproducir fielmente el comportamiento de los elementos presentes en una escena. El papel de estos motores es esencial para que el trabajo final muestre un resultado realista.

El objetivo de este proyecto es el desarrollo e implementación de un motor físico multifuncional, modular y ampliable que pueda ser usado en diferentes campos, como los videojuegos y la simulación de sistemas físicos.

Este proyecto implementa los conceptos de dinámica de sólidos rígidos y dinámica de partículas. Además se han introducido otras funcionalidades derivadas del tratamiento de estos cuerpos como son las colisiones de sólidos rígidos y las uniones elásticas.

El motor se realizará sobre el lenguaje C++, siendo el lenguaje más común en la realización de este tipo de aplicaciones y permitiendo así su fácil acoplamiento en los campos anteriormente mencionados.

Abstract.

Both the video game industry such as animation industry requires physics engines that allow to faithfully reproducing the behavior of the elements present in a scene. The role of these motors is essential for the final work to show a realistic outcome.

The aim of this project is the development and implementation of a multi-physics engine, modular and expandable, that can be used in different fields, such as video games and simulation of physical systems.

This project implements the concepts of rigid bodies and particle dynamics. We have also introduced other features resulting from the processing of such bodies as are collisions of rigid bodies and elastic joints.

The engine will be made on the language C++, being the most common language in this type of application, thus allowing easy coupling in the fields mentioned above.

Palabras clave: sólido rígido, partícula, motor física, colisiones, unión, videojuegos, simulación, animación, OpenGL, Euler.

Contenido

1. Introducción.

Un motor de física se puede definir como un software informático que permite simular sistemas físicos como sólidos rígidos, fluidos y sistemas de partículas entre otros.

El proyecto propone implementar un motor físico que pueda ser empleado en el desarrollo de simuladores, videojuegos u otras aplicaciones que requieran el cálculo de sistemas físicos.

1.1. Desarrollo de los motores de física.

La evolución de estos motores ha ido de la mano con la de los videojuegos. Se puede considerar el inicio de la física en videojuegos en videojuegos antiguos como PONG (1972) u otros posteriores en los que simplemente se calculaba la trayectoria de un proyectil con la única restricción de la gravedad. A medida que los consumidores demandaban más realismo, estos motores debían ser más precisos. Una mejora gráfica y sonora quedaría deslucida sin un consecuente comportamiento físico realista del entorno.

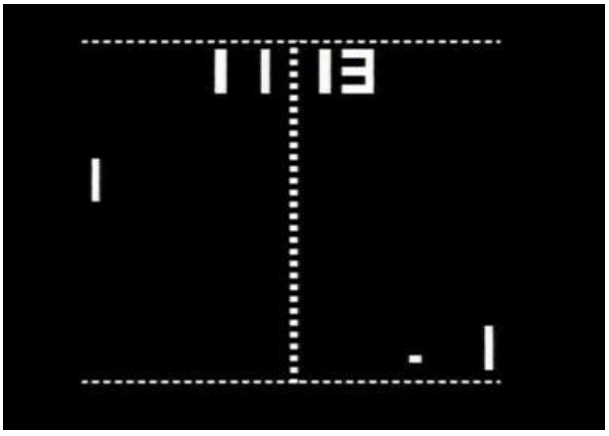


Figura 1.1 *Pong (1972)*

La principal traba en la evolución tanto de la calidad gráfica así como del comportamiento físico realista ha sido el manejo de recursos. En un principio renderizado, comportamiento físico, sonido… recaían en la CPU. Con la mejora de los

gráficos la CPU se veía saturada, por lo que una solución sería liberar a esta de los procesos gráficos, estrategia utilizada posteriormente para otros aspectos de la aplicación. Para satisfacer la demanda de los consumidores de una mayor calidad se utilizaron las tarjetas graficas llamadas *aceleradores gráficos* para liberar de esta carga a la CPU. Las generaciones sucesivas de estas tarjetas fueron las tarjetas 3D. Con el aumento de la complejidad de modelos así como de las tasas de renderizado vino la evolución de estas tarjetas gráficas. La última generación de estas tarjetas incorporó un modulo llamado GPU (Graphic Processing Unit) que manejaba todos los procesos relacionados con los gráficos. Una mejora en estas tarjetas llegó en 1997 con el lanzamiento al mercado de la tarjeta Voodoo 1 de la compañía 3Dfx.

Figura 1.2 *Voodoo 1 de la compañía 3Dfx.*

Con la evolución de las CPU´s y las GPU´s, la tecnología fue suficientemente capaz como para proporcionar un entorno convincente. Pero la carrera por la simulación no debía acabarse ahí. Adelantándose a los acontecimientos, una compañía llamada MathEngine empezó a desarrollar lo que llamo un *motor físico.* En principio el resultado fue bueno, pero mostraba irregularidades como la no estabilización de los objetos en la superficie. La estabilidad es un requisito importante, por no decir esencial, en un motor, pero también lo es la robustez. Otra compañía que lanzó un producto de gran calidad por entonces fue Telekinesis, llamando a su motor Havok. Con el tiempo este motor se convertiría en uno de los mejores del mercado, como se reseñará más adelante en el estudio de motores comerciales en la actualidad.

Actualmente la mejora tecnológica de las CPU´s y tarjetas gráficas ha traído consigo la consecución de un realismo enorme. Pero otro factor ha ayudado sustancialmente en ello: la integración de PPU´s (Physics Processing Unit) en las tarjetas gráficas. Siguiendo con la estrategia de liberar al procesador de cálculos para

conseguir mayor rendimiento, estos módulos concentran gran parte de los cálculos físicos realizados en las aplicaciones. Ageia en su motor Physx ha sido uno de los primeros en incorporar este dispositivo en sus tarjetas.

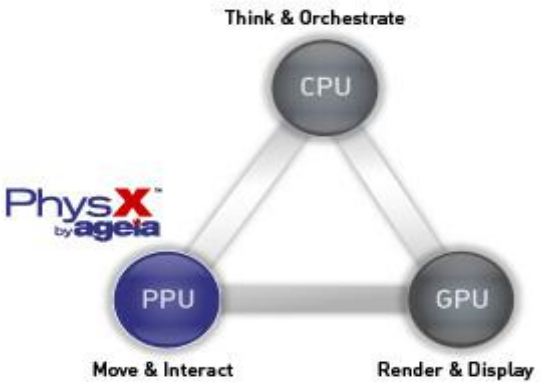


Figura 1.3 *Esquema de distribución de tareas.*

Como conclusión se advierte que la evolución de los motores de física ha ido ligada, así como a su propio desarrollo, a la tecnología que los soporta. Esta evolución esta inevitablemente ligada a la demanda del usuario de mayor realismo en los entornos, que se aplica no sólo en los videojuegos, sino también en otras industrias como la animación y la simulación.

1.2. Función de un motor de Física.

La funcionalidad de un motor de física se puede explicar como la de hacer los cálculos correspondientes para dar como resultado un comportamiento real en el entorno en el que se le proponen dichos cálculos. Además de hacer ecuaciones diferenciales que prevean el comportamiento de sistemas físicos, debe manejar el comportamiento independientemente del entorno, es decir, que el código pueda ser reusable en diferentes situaciones y, dados unos parámetros, manejar su comportamiento.

Para ello debe manejar conceptos físicos como cinemática y dinámica de sólidos rígidos y blandos, manejo de colisiones, fluidos, integración de ecuaciones diferenciales y otros conceptos.

1.3. Objetivos.

El objetivo principal del proyecto es la construcción de un Motor de Física multifuncional, sobre el cual simularemos la dinámica de las partículas, así como el comportamiento de los sólidos rígidos, y su interacción con el entorno.

Además del núcleo del Motor, se debe generar una API que facilite su uso en aplicaciones externas y permita visualizarlo de manera sencilla.

Los principales problemas que debemos resolver en la implementación son:

1. Dinámica de partículas. El sistema debe permitir la simulación de sistemas de partículas, que permitirán al usuario final crear diversos efectos.
2. Dinámica de Sólidos Rígidos. El sistema debe responder de manera realista a las distintas fuerzas externas que actúen sobre los distintos sólidos creados en el mundo durante la simulación.
3. Dinámica de Colisiones. Se debe intentar que el sistema responda de manera eficaz y realista a los choques generados por la interacción de los sólidos con el entorno.
4. Uniones elásticas entre sólidos. El sistema debe permitir la unión de sólidos mediante tensores y simular su comportamiento físico.

Para comprobar el correcto funcionamiento del motor se ha de crear una batería de pruebas visibles sobre la plataforma gráfica OpenGL.

Toda la implementación será sobre la CPU, sin el empleo de GPU´s, con lo cual se conseguirá que esta herramienta pueda ser utilizada ampliamente en cualquier ordenador personal. El lenguaje de programación será en C++, sobre la utilidad Visual Studio 2008.

1.4. Planificación.

El desarrollo de este proyecto empezó en el verano de 2009, con la impartición de tutorías que nos mostraban los fundamentos a seguir en el desarrollo de este proyecto. La planificación que se propuso se cumplió (en mayor o menor medida) siguiendo el siguiente esquema, significando cada número uno de los componentes del grupo:

1.4.1. Tareas a desarrollar.

1. Realizar un estudio del estado actual de los motores de física comerciales.
2. Proponer una arquitectura para el Motor y sus APIs.
3. Implementación del Núcleo del Motor.
4. Implementación del Visualizador.
5. Implementación del Demostrador.
6. Documentación.
7. Presentación.

Tar	Ago	Sep	Oct	Nov	Dic	Ene	Feb	Mar	Abr	May	Jun	Jul
1	123											
2			123									
3				123	123	123						
4								12				
5								3	123			
6			123	123		123		123	123	123		
7												123

2. Estado del arte y motores comerciales.

En la actualidad existen múltiples motores de física, destinados en su mayoría a los videojuegos o a la simulación. Se ha centrado este estudio en tres motores pretendiendo ejemplificar el estado actual de este mercado. Estos motores son ODE, PhysX y, por último, Havok.

Physx y Havok al ser motores privados no facilitan su implementación interna, aunque proporcionan gratuitamente herramientas que permiten construir ejemplos para ver sus funcionalidades. ODE en cambio al ser un motor de código abierto facilita su implementación interna así como una completa guía de usuario que además de incluir instrucciones sobre su funcionamiento e instalación contiene nociones teóricas y prácticas sobre los conceptos utilizados en este motor, como por ejemplo sólidos rígidos, uniones y su integrador.

Por ello nuestro proyecto se basa en una estructura parecida a la de ODE, por simpleza y recursos biográficos. Además se hace un estudio más exhaustivo que permita comprender mejor ODE y, por lo tanto, explicar ciertos fundamentos comunes del nuestro.

2.1. ODE.

ODE (Open Dynamics Engine) es una librería de software libre, de calidad profesional, dirigida a la simulación de cuerpos rígidos articulados. Las aplicaciones probadas sobre este motor incluyen la simulación de vehículos, criaturas bípedas y demás cuerpos sobre un entorno de realidad virtual. Entre sus características también incluye la detección de colisiones integrada.

Su autor es Russell Smith, con la ayuda de diferentes colaboradores, dando comienzo el proyecto en 2001.

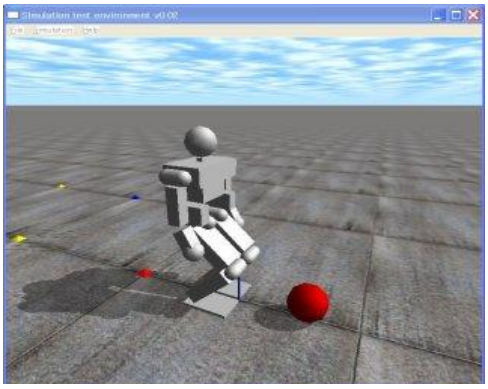


Figura 2.1 *Simulación sobre ODE.*

Al ser una librería de software libre, hay gran cantidad de bancos de pruebas y manuales, estando en continuo proceso de evolución y mejora.

2.1.1. Características.

ODE está enfocado a la simulación tanto en tiempo real como interactiva de estructuras articuladas, las cuales consisten en cuerpos rígidos de diferente forma conectados entre sí mediante conexiones de diferentes tipos. Se incluyen gran cantidad de conexiones, pudiendo manejar una extensa variedad de giros y desplazamientos entre los cuerpos unidos.

Este motor prioriza velocidad y estabilidad a costa de precisión física, lo cual lo hace robusto y fiable, pero también poco preciso respecto a otros motores del mercado. Pero la ventaja es la gran capacidad de personalización del entorno (colisiones, etc.…) incluso durante la simulación sin la aparición de errores.

Respecto al tratamiento de colisiones, ODE utiliza colisiones duras, lo cual quiere decir que hay impuestas restricciones especiales cuando dos cuerpos entran en contacto. En otros motores se utilizan otras alternativas, siendo sin embargo más complicadas y más difíciles de implementar libres de errores.

La interfaz original fue implementada en C, aunque está escrito mayoritariamente en C++ y se ha implementado una interfaz en C++ sobre la original.

2.1.2. Conceptos.

Características de los sólidos rígidos.

Los sólidos rígidos tienen varias características desde el punto de vista de la simulación. Algunas de estas cambian durante la simulación:

- Vector de posición(x,y,z) del punto de referencia del cuerpo. En ODE el punto de referencia corresponde con el centro de masas.
- Velocidad lineal del punto de referencia, representada por un vector (vx,vy,vz).
- Orientación: representada por un quaternion(qs,qx,qy,qz) o una matriz de dimensión 3x3.
- Vector de velocidad angular (wx,wy,wz) que describe como cambia la orientación a lo largo del tiempo.

Otras propiedades permanecen constantes:

- Masa.
- Posición del centro de masas respecto al punto de referencia, en ODE estos dos puntos coinciden.
- Matriz de inercia: matriz de 3x3 que describe como la masa está distribuida alrededor del centro de masas.

Conceptualmente cada cuerpo tiene un eje de coordenadas dentro de, que se mueve junto al cuerpo, como se muestra en la figura. El origen de este eje es el punto de referencia. En ODE algunos valores tales como vectores y matrices utilizan estos ejes de coordenadas y otros utilizan el eje de coordenadas global.

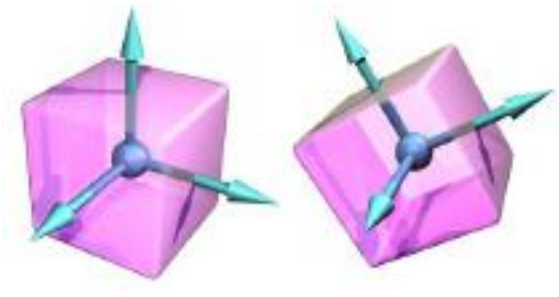


Figura 2.2 *Eje de coordenadas de los cuerpos.*

Un hecho importante es que la forma del objeto no es una propiedad dinámica. La detección de colisiones es la parte del motor que se ocupa de manera detallada de ello.

Islas y cuerpos desactivados.

Los cuerpos están conectados unos a otros por uniones. Una "isla" es un grupo de cuerpos que no pueden ser separados. En otras palabras, cada cuerpo está conectado de alguna manera con cada otro cuerpo de la isla.

Cada isla es tratada por separado en cada paso de simulación, por lo que si hay N islas similares, el coste de cada paso será O(N). Una manera de disminuir este coste sería desactivando cuerpos irrelevantes para ciertos pasos de la simulación, ya que estos no son actualizados durante estos pasos.

Integración

El proceso que simula el sistema de sólidos rígidos a lo largo del tiempo se llama integración. Con cada paso de integración se avanza desde el tiempo actual al tiempo correspondiente, siendo este la suma del actual mas el paso de integración dado. El motor calcula el estado correspondiente de los cuerpos en ese momento. Por lo que hay que considerar:

- Precisión: cuanto se aproxima el motor a la "vida real".
- Estabilidad: cuanto aguantara el motor sin producir errores de cálculo.

ODE es un motor estable, pero no muy preciso, a menos que el paso de integración sea pequeño. Aunque, excepto para simulaciones de alto nivel, este motor parece físicamente perfecto.

Acumuladores de fuerzas.

Entre paso y paso, el usuario puede llamar a funciones que apliquen fuerzas a los cuerpos, sumándose estas en los acumuladores de fuerzas. En el siguiente paso de simulación estas serán consideradas, y los movimientos se harán consecuentemente. Después de cada paso de integración los acumuladores de fuerzas son puestos a cero.

Uniones.

Las uniones, como su propio nombre indica, sirven para conectar dos cuerpos. Se pueden entender también como la relación entre las posiciones de dos cuerpos, definiendo esta relación las posiciones y orientaciones posibles entre los dos. Tres ejemplos de uniones son:

- "Bola" y "cuenca": la "bola" del primer cuerpo tiene que ir unida al engranaje del segundo objeto, permitiendo un movimiento más o menos "orbital" alrededor de la unión.
- Bisagra: esta unión permite un movimiento similar al de una bisagra de una puerta.
- Pistón: los cuerpos deben estar alineados, además de tener la misma orientación.

Figura 2.3 *Tres tipos de uniones: bola, bisagra y pistón.*

Grupos de uniones.

Los grupos de uniones son contenedores que almacenan a estas. Las uniones pueden ser añadidas a un grupo de una en una y ser destruidas con una llamada a una función. Pero no pueden ser destruidas de una en una, tienen que ser destruidas simultáneamente.

Esto es muy útil para las uniones de contacto, que son añadidas y quitadas del mundo en cada paso de simulación.

Error de unión y parámetro de reducción de error(ERP).

Cuando se produce una unión, los cuerpos deben estar uno con respecto al otro de tal manera que la unión permita esta situación. Pero puede ser que esto no sea así. Esto puede suceder por dos causas:

- El usuario coloca un cuerpo sin tener en cuenta la posición/orientación del otro.
- Por errores surgidos durante la simulación, que los apartan de las posiciones deseadas.

Existe un mecanismo para reducir este tipo de situaciones: durante cada paso de simulación cada unión aplica una fuerza especial para colocar los cuerpos en la posición y/o orientación deseada.

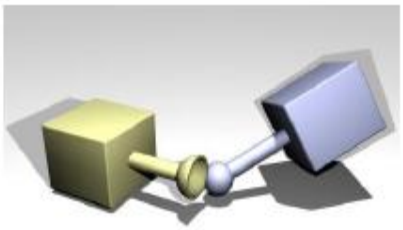


Figura 2.4 *Ejemplo de error en la unión.*

Esta fuerza está controlada por el parámetro ERP, cuyo valor puede oscilar entre 0 y 1.

El ERP indica que proporción de errores de unión va a ser arreglada durante el siguiente paso de simulación. Si este valor es 0 no se corregirá ningún error, mientras que si el valor es 1 se corregirán todas las uniones erróneas. Un valor de 1 tampoco es recomendado, debido a que no todas las uniones se corregirán debido a aproximaciones internas. Un valor entre 0.1 y 0.8 es el recomendado (0.2 por defecto).

Un valor global de ERP puede ser introducido de tal manera que afecte a la mayoría de uniones, pero también se puede introducir un valor local para que controle solo a ciertas uniones.

Restricciones suaves.

Muchas restricciones son por naturaleza “duras”. Esto significa que estas restricciones nunca son violadas. En la práctica estas restricciones pueden ser violadas

por errores no intencionados introducidos en el sistema, pero el ERP puede ser configurado para solventarlos.

No todas las restricciones son “duras”, algunas restricciones “blandas” son asignadas para poder ser violadas. Por ejemplo, las restricciones de contacto asignadas a las colisiones son duras, como si los objetos fueran de metal, pero pueden ser configuradas como blandas para simular materiales más blandos, permitiendo algo de penetración entre estos cuerpos cuando entran en contacto.

Esta distinción entre restricciones duras y blandas es controlada mediante dos parámetros: el ERP mencionado anteriormente y el Constraint Force Mixing(CFM), descrito a continuación.

A continuación se expone una descripción técnica del significado del CFM. Usualmente la ecuación de la restricción para cualquier unión tiene la forma:

$$J * v = c$$

Donde v es el vector de velocidad de los cuerpos envueltos, J es una matriz Jacobiana con una fila para cada grado de libertad que la matriz quita del sistema, y c es un vector de restricciones. En el siguiente paso, un vector lambda es calculado (del mismo tamaño que c) con lo que las fuerzas aplicadas a los cuerpos son

$$fuerza = JT * \lambda$$

ODE le da una nueva perspectiva. La ecuación de restricción de ODE tiene la forma:

$$J * v = c + CFM * \lambda$$

Donde CFM es una matriz cuadrada diagonal. El CFM mezcla la fuerza restrictiva resultante con la restricción que la produce. Un valor no nulo(positivo) del CFM permite que la ecuación de restricción original sea violada por una cantidad

proporcional a CFM veces la fuerza reparadora λ que es necesaria para restaurar la restricción. Solucionando para λ:

$$(JM - 1JT + CFM/h)\ \lambda = c/h$$

De esta manera CFM simplemente se añade a la diagonal de la matriz del sistema original. Usar un valor positivo del CFM tiene el beneficio adicional de evitar al sistema ninguna particularidad y así mejorar la precisión de la factorización.

¿Cómo usar ERP y CFM?

ERP y CFM pueden ser independientemente configuradas en muchas uniones, como en uniones de contacto, limites de uniones, etc. Para controlar las características de la unión(o el límite).

Con el valor de CFM a cero, la restricción será dura. En cambio con valor positivo será posible violar la restricción. En otras palabras, la restricción será blanda en proporción a CFM, esta podrá ser violada una cantidad proporcional a CFM veces la fuerza restauradora que es necesaria para restaurar la restricción. Con valores negativos de CFM se obtendrían resultados inesperados.

Con valores de CFM y ERP apropiados se pueden obtener varios efectos, tales como esponjosidad, oscilación, etc. De hecho ERP y CFM pueden ser configuradas para tener el mismo efecto que cualquier muelle con el "damping deseado". Si se tiene una constante de muelle k_p y una constante de damping k_d, entonces las constantes correspondientes de ODE son:

$$ERP = hkp/\ (hkp + kd)$$
$$CFM = 1\ /\ (hkp + kd)$$

Donde h es el tamaño del paso. Estos valores darán el mismo efecto que un sistema con muelles y damping, con integración de primera orden implícita. Incrementando CFM , especialmente su valor global, reduciría errores numéricos.

Manejo de colisiones.

Las colisiones entre cuerpos entre sí mismos o entre el entorno estático son manejadas como se explica a continuación:

- Antes de cada paso de simulación, el usuario llama a las funciones de detección de colisiones para determinar que cuerpos están en contacto, devolviendo una lista de puntos de contacto. Cada punto de contacto especifica una posición en el espacio, una normal de superficie y la profundidad de penetración.
- Una unión especial de contacto es creada para cada punto de contacto. A la unión se le proporciona información acerca del contacto, como por ejemplo la fricción presente en la superficie de contacto, como es de suave, etc.
- Las uniones de contacto son recogidas en un grupo de uniones, que les permite ser añadidas y quitadas del sistema rápidamente. La velocidad de simulación decrece a medida que el número de contactos crece, pero existen varias estrategias para disminuir el número de uniones.

Código de simulación típico.

Este código procedería de la siguiente forma:

- Creación del mundo.
- Creación de cuerpos en el mundo.
- Establecer estado (posición, etc.…) de todos los cuerpos.
- Crear uniones en el mundo.
- Conectar las uniones a los cuerpos.
- Fijar los parámetros de todas las uniones.
- Crear el mundo de colisiones y la geometría de colisiones, si es necesario.
- Crear un grupo de uniones para contener las uniones de contacto.
- Bucle:
    - Aplicar fuerzas a los cuerpos si es necesario.
    - Ajustar los parámetros de las uniones si es necesario.
    - Llamar a la detección de colisiones.
    - Crear una unión de contacto para cada punto de colisión, y poner a esta unión en el grupo de unión de contacto.
    - Dar un paso de simulación.
    - Quitar todas las uniones en el grupo de uniones de contacto.
    - Destruir el mundo dinámico y de colisiones.

2.1.3. Juegos en los que se usa.

Ha sido utilizado en diferentes videojuegos de distintas plataformas. Algunos de los videojuegos que han utilizado la física de ODE:

- BloodRayne 2
- Call of Juarez
- Call of Juarez: Bound in Blood
- S.T.A.L.K.E.R and Clear Sky
- Resident Evil 4
- Resident Evil: The Umbrella Chronicles
- World of Goo

Figura 2.5: *World of Goo* ***y*** *Call of Juarez*

2.2. Physx.

La PPU (Unidad de Procesamientos de Física) **PhysX** es un chip y un kit de desarrollo diseñados para llevar a cabo cálculos físicos muy complejos. Conocido anteriormente como la SDK de NovodeX, fue originalmente diseñada por AGEIA y tras la adquisición de AGEIA, es actualmente desarrollado por Nvidia e integrado en sus chips gráficos más recientes.

El 20 de julio de 2005 Sony firmó un acuerdo con AGEIA para usar la SDK de Novodex en la consola Playstation 3. Esto provocó que muchos desarrolladores empezaran a crear juegos muy complejos gracias a esta tecnología. AGEIA afirmó que el PhysX era capaz de realizar estos procesos de cálculos físicos cien veces mejor que cualquier CPU creada anteriormente.

Figura 2.6 *PPU Physx.*

2.2.1. Características.

Las características incluyen:

Complex rigid body object physics system

La componente dinámica de los cuerpos rígidos le permite simular objetos con un alto grado de realismo. Hace uso de conceptos físicos como marcos de referencia, posición, velocidad, aceleración, movimiento, fuerzas, el movimiento de rotación, la energía, la fricción, el impulso, las colisiones, las limitaciones, y así sucesivamente con el fin de darle un kit de construcción con los que se pueden construir una amplia variedad de tipos de dispositivos mecánicos. Esta característica incluye:

- Colisiones primitivas (esfera, caja, cápsula)
- Varios tipos de articulaciones (esférica, de revolución, prismáticos, cilindros, fijo)
- Ragdoll avanzada, con las articulaciones. Creación y edición.
  Materiales y el modelo de fricción en la superficie de apoyo de la colisión.
- Colisión de apoyo continuo para una rápida detección de objetos en movimiento.

Ray-cast y la dinámica del vehículo articulado.

NVIDIA PhysX SDK tiene la capacidad de simular los vehículos que nos encontramos en la vida real:

- Coches Raycast - cuerpo rígido
- Rueda de formas esféricas - sofisticado modelo de fricción del neumático

- Unión basada en la suspensión - dirección y suspensión de geometrías complejas

Multi-threaded/Multi-platform/PPU habilitado

PhysX está optimizada para el PC de un solo núcleo y multi-núcleo, Xbox 360, PLAYSTATION 3.

Reacción de líquidos y su simulación

Los fluidos permiten la simulación de líquidos y gases mediante un sistema de partículas y emisores con la posibilidad de que el usuario pueda controlar la simulación de cada uno. También permite la simulación de explosiones y efectos de los desechos

Simulación de tejidos.

La característica de tela del SDK PhysX de NVIDIA permite la simulación de artículos hechos de tela, tales como banderas, ropa, etc.

Cuerpos Blandos

La función de cuerpo blando del SDK PhysX de NVIDIA permite la simulación de objetos deformables. También se puede utilizar para artículos que no se suelen considerar como cuerpos blandos, tales como las plantas o varias capas de tela.

Simulación de campos de fuerzas

Los campos de fuerza son objetos SDK similar a los actores, que afectan a los tejidos, elementos de cuerpo suave, fluido y rígido que entran en su área de influencia. Estos campos de fuerza permiten poner en práctica las ráfagas de viento, aspiradoras o zonas con fuerzas gravitatoria.

2.2.2. Juegos en los que se usa.

- 2 Days to Vegas
- Adrenalin 2: Rush Hour
- Age of Empires III
- Age of Empires III: The WarChiefs

- Alpha Prime
- Auto Assault
- B.A.S.E. Jumping
- Bet on Soldier: Blackout Saigon
- Bet on Soldier: Blood of Sahara
- Bet on Soldier: Blood Sport
- Cellfactor: Combat Training
- Cellfactor: Revolution
- City of Villains
- Crazy Machines II
- Cryostasis
- Desert Diner
- Dragonshard
- Dusk 12
- Empire Above All
- Empire Earth III
- Entropia Universe
- Fallen Earth
- Fury
- Gears Of War
- Gluk'Oza: Action
- GooBall
- Gothic 3
- Gunship Apocalypse
- Heavy Rain
- Hero's Jorney
- Hour of Victory
- Hunt, The
- Huxley
- Infernal
- Inhabited island: Prisoner of Power
- Joint Task Force
- Magic ball 3
- Mass Effect
- Medal of Honor: Airborne
- Metro 2033
- Mobile Suit Gundam: Crossfire
- Monster Madness: Battle for Suburbia
- Monster Truck Maniax

- Myst Online: URU Live
- Red Steel
- Rise Of Nations: Rise Of Legends
- Roboblitz
- Sacred 2
- Sherlock Holmes: The Awakened
- Showdown: Scorpion
- Silverfall
- Sovereign Symphony
- Sonic and the Secret Rings
- Speedball 2
- Stalin Subway, The
- Stoked Rider: Alaska Alien
- Switchball
- Tension
- Tom Clancy's Ghost Recon Advanced Warfighter
- Tom Clancy's Ghost Recon Advanced Warfighter 2
- Tom Clancy's Rainbow Six Vegas
- Tom Clancy's Splinter Cell: Double Agent (multiplayer)
- Tortuga: Two Treasures
- Two Worlds
- Ultra Tubes
- Unreal Tournament 3
- Warfare
- Warmonger: Operation Downtown Destruction
- W.E.L.L. Online
- Winterheart's Guild

Figura 2.8 *Unreal Tournament 3*

2.3. Havok.

Havok Game Dynamics SDK es un motor físico (simulación dinámica) utilizada en videojuegos que recrea las interacciones entre objetos y personajes del juego. Por lo que detecta colisiones, gravedad, masa y velocidad en tiempo real llegando a recrear ambientes realistas y naturales.

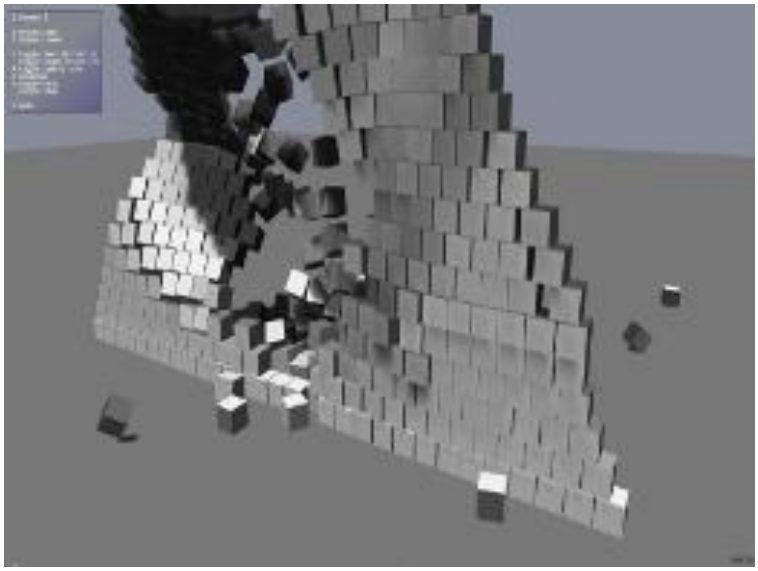


Figura 2.9 *Simulación sobre Havok.*

El motor Havok fue comprado por Intel para hacer competencia a su principal rival, Physx.

2.3.1. Características.

En sus últimas versiones se ejecuta por entero por hardware mediante el uso de la GPU, liberando así de dichos cálculos a la CPU. Este se apoya en las librerías Direct3D y OpenGL compatibles con ShaderModel 3.0.

Havok Physics ofrece un gran rendimiento en el tiempo de detección de colisiones y simulación real de soluciones físicas. Ofrece rapidez y robustez por lo que se ha convertido en referencia dentro de la industria de los videojuegos. También ha sido elegido por los principales desarrolladores de juegos en más de 200 títulos en marcha y otros en desarrollo.

Continuous Physics

La robustez es la exigencia más importante de este motor. Utilizando técnicas innovadoras propiedad de la empresa, ofrece una física sin fallos que le permite eliminar las restricciones de diseño imponer otras soluciones.

Multiplataforma

Es ampliamente utilizado en todos los juegos de plataformas de hoy en día: Microsoft Xbox 360 , Sony PlayStation 3, Nintendo Wii , Microsoft Xbox ® PlayStation 2, PSP y PC.

Battle Tested (Testeo de usuario)

La documentación de ayuda, herramientas de creación de contenidos, depuración y personalización han hecho de este motor un motor de referencia en este mercado. Todo ello sumado a los informes de los consumidores han conseguido hacer una herramienta madura y de confianza para los desarrolladores de video juegos.

Flexible y fácil de usar

Havok Physics se ha utilizado en todos los géneros de juegos, y ha evolucionado para satisfacer las necesidades de la mayoría de los equipos exigentes de la industria. Se integra fácilmente en cualquier proceso de desarrollo de juegos, tanto comerciales como domésticos. Basado exclusivamente en las necesidades del cliente, sólo se refiere a las plataformas hardware que los clientes necesitan, y como resultado no impone las rigideces innecesarias y es totalmente abierto en su arquitectura.

Destacados

Havok Physics proporciona un completo apoyo a las siguientes áreas:

- Detección de colisiones.
- Dinámica y solución de restricciones.
- Control de personajes.
- Herramientas de creación de contenido para 3ds Max ®, Maya y XSI ®.
- Depurador visual y de memoria así como herramientas de rendimiento.
- Solución integrada de vehículos.
- SDK de reflexión y la materialización de clases.

Havok Physics es totalmente multiproceso y multi-plataforma.

2.3.2. Juegos en los que se usa.

- Age of Empires III
- Alone in the dark 5
- Amped 3
- Armed and Dangerous
- Assassin's Creed
- BioShock
- Brute Force
- Call of Duty 4: Modern Warfare
- Company of Heroes
- Counter-Strike: Source
- Crackdown
- Crash Nitro Kart
- Day of Defeat: Source
- Dead Rising
- Demon's souls
- Destroy All Humans!
- Deus Ex: Invisible War
- Diablo III
- Disaster: Day of Crisis
- Evil Dead Regeneration
- F.E.A.R.
- MadWorld
- From Russia with Love
- Full Spectrum Warrior
- Grand Theft Auto IV
- Half-Life: Source
- Half-Life Deathmatch: Source
- Halo 2
- Halo 3
- Max Payne 2: The Fall of Max Payne
- Medal of Honor: Pacific Assault
- Mercenaries: Playground of Destruction
- Middle-earth Online
- Motorstorm
- Painkiller
- Perfect Dark Zero

- Pitfall: The Lost Expedition
- Psi-Ops: The Mindgate Conspiracy
- Red Faction 2
- Robotech: Invasion
- Resident Evil 5
- Resident Evil: The Darkside Chronicles
- Silent Hill 5
- Sonic the Hedgehog (2006 next-generation game)
- Sonic Unleashed
- Spider-Man 2
- StarCraft II
- Star Wars: The Force Unleashed
- Super Smash Bros. Brawl
- Swat 4
- Test Drive Unlimited
- The Matrix: Path of Neo
- The Punisher
- Thief: Deadly Shadows
- Top Spin 3
- Tom Clancy's Ghost Recon 2
- Torque: Savage Roads
- Tribes: Vengeance
- Uru: Ages Beyond Myst
- WWE Crush Hour
- Syphon Filter Dark Mirror
- Syphon Filter Logan's Shadow
- Killzone Liberation
- Killzone 2
- WWE SmackDown vs. Raw 2010
- Halo 3: ODST

3. Fundamentos físicos teóricos.

Para poder entender de manera correcta el comportamiento de los objetos y sobre todo su interacción física con el entorno, es necesario introducir algunos conceptos físicos sobre los que hemos basado nuestra implementación.

A continuación se explican ciertos conceptos que se utilizan en el motor. Primero se explicarán las partículas para luego introducir los sólidos rígidos, que se pueden considerar como un caso complejo de las primeras. Luego se explicaran otros dos conceptos que nos permitirán actuar con los sólidos rígidos y las partículas como son los springs y las colisiones.

3.1. Dinámica de partículas.

Las partículas son objetos que tienen masa, posición, velocidad y respuesta a las fuerzas, pero no ocupan un volumen en el espacio. Ya que son simples, las partículas son con diferencia los objetos más fáciles de simular. A pesar de su simplicidad, pueden exhibir una gran variedad de comportamientos interesantes. Por ejemplo, una amplia variedad de estructuras no rígidas pueden ser construidas con la conexión entre partículas.

El movimiento de una partícula está gobernado por la formula $f = ma$ o, escrito de otra manera, $x'' = f/m$. El problema de esta ecuación deriva en que contiene a una segunda derivada, por lo que se convierte en una ecuación diferencial de segundo orden. Para poder manejar este concepto, lo convertimos en una ecuación de primer orden introduciendo variables adicionales. Se crea una variable v para representar la velocidad, dándonos dos ecuaciones diferenciales, $v' = f/m$ y $x' = v$.

3.1.1. Sistemas de partículas básicos.

Una simulación de partículas incluye dos partes principales: las partículas en sí mismas y las entidades que aplican fuerzas a estas. Vamos a considerar las partículas sin fuerzas aplicadas a ellas primero, para poder explicar cómo representarlas.

Las partículas constan de masa, posición, velocidad y además pueden ser objeto de fuerzas. Para poder manejar los sistemas de partículas se deben aplicar los mismos conceptos de dinámica que a los sólidos rígidos, únicamente teniendo en cuenta que estas no tienen forma ni volumen por lo que se descartan conceptos como rotación, matriz de inercia y demás conceptos relacionados con el volumen.

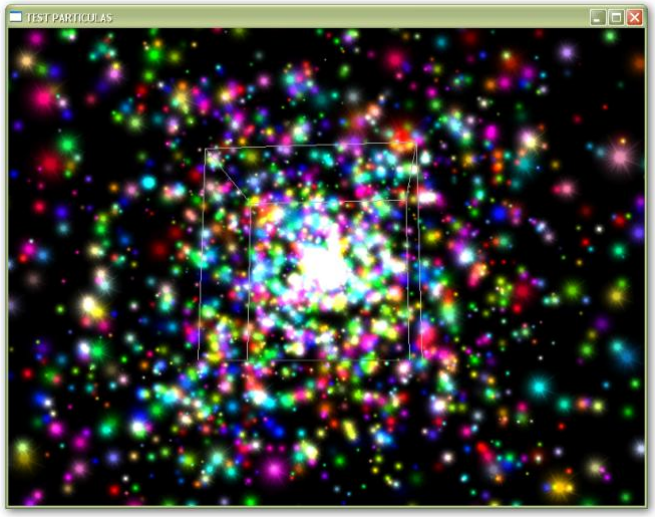


Figura 3.1: *Ejemplo de sistema de partículas*

3.1.2. Fuerzas.

Se pueden considerar a todas las partículas iguales al contrario que a los objetos que al estar influenciados por fuerzas externas reaccionan dependiendo de su forma y masa.

$$X$$
$$V$$
$$f$$
$$m$$

Figura 3.2: *Estructura de las partículas.*

Siendo X la posición de la partícula, V la velocidad (cuya unión daría como resultado el espacio de fase), f el acumulador de fuerzas y m la masa.

Las fuerzas se pueden clasificar en tres tipos:

- Fuerzas unarias, como la gravedad, que actúan independientemente en cada partícula, ejerciendo sobre ellas una fuerza constante, o una fuerza que depende de la posición, velocidad o tiempo de una o más partículas.
- Fuerzas n-arias, como las uniones: aplican fuerzas a un conjunto concreto de partículas.
- Fuerzas de interacción espacial, como la atracción y la repulsión: pueden actuar en cualquier conjunto de partículas, dependiendo de su posición.

3.1.2.1. Fuerzas unarias.

El ejemplo mas claro es la gravedad, que se rige por $f = mg$, donde g es un vector, cuya magnitud es la constante gravitacional. Si todas las partículas están sujetas a esta fuerza, para simularla se recorren las partículas añadiendo al acumulador de fuerzas de cada una dicha fuerza.

3.1.2.2 .Fuerzas n-arias.

Un ejemplo de fuerza n-aria, siendo n= 2, es la ley de muelles de Hooke. En una simulación básica de masa y muelle, los muelles son el elemento estructural que mantiene todo unido. Las fuerzas entre dos partículas en posiciones a y b son:

$$f_a = -\left[k_s(|l| - r) + k_d \frac{l' * l}{|l|}\right]\frac{l}{|l|}, \qquad f_b = -f_a$$

Donde f_a y f_b son las fuerzas sobre a y b, respectivamente, $l = a - b$, r es la distancia en reposo, k_s es la constante elástica del muelle y, para solucionar el problema que haría que estas uniones oscilaran infinitamente, la constante de Damping k_d. l' es la derivada de l, que es la diferencia entre velocidades $v_a - v_b$ de las dos partículas.

En la ecuación anterior la fuerza del muelle es proporcional a la diferencia entre la longitud actual y la longitud en reposo, mientras que k_d es proporcional a la

velocidad de acercamiento entre a y b. Fuerzas iguales y opuestas actúan en cada partícula, así como el muelle que las une.

3.1.2.3. Fuerzas de interacción espacial.

Un muelle aplica fuerzas a dos partículas en concreto. En cambio, las fuerzas de interacción espacial pueden actuar en cualquier conjunto (pudiendo ser este conjunto de cualquier tamaño) de partículas. Para fuerzas de interacción local, las partículas empiezan a interactuar cuando se acercan y dejan de hacerlo cuando se alejan. Estos modelos de interacción entre partículas se pueden usar para simular fluidos.

3.2 . Dinámica de sólido rígido.

Se define como solido rígido a una idealización de de los sólidos, es decir, cuerpos en los que se ignoran las deformaciones y en los que la distancia entre dos puntos cualquiera del cuerpo permanecen constantes.

3.2.1. Posición y orientación.

La localización de una partícula en el espacio en un instante t puede describirse como un vector $x(t)$, que describe la traslación de una partícula desde el origen. Los sólidos rígidos son más complicados, ya que además hay que considerar su rotación. Para localizar un sólido rígido en el espacio, se debe usar un vector $x(t)$, que describe la traslación del cuerpo, también se debe describir la rotación del cuerpo, usando una matriz de rotación de 3x3 ($R(t)$). Estas son las llamadas variables espaciales de un cuerpo rígido.

Un sólido rígido, a diferencia de una partícula, ocupa un volumen de espacio y tiene una forma particular. Ya que en un sólido rígido se tiene rotación y traslación, la forma de este, se define como un espacio constante llamado espacio del cuerpo.

Dada una descripción geométrica del cuerpo en el espacio del cuerpo, se usa $x(t)$ y $R(t)$ para transformar el espacio del cuerpo en el espacio del mundo. Para simplificar ciertas ecuaciones, se requerirá que nuestra descripción del solido rígido en

el espacio del cuerpo será tal que el centro de masas del cuerpo estará emplazado en el origen $(0,0,0)$. Se puede considerar que el centro de masa coincide con el centro geométrico del cuerpo. Para describir la forma del cuerpo requerimos que este centro geométrico este en el punto $(0,0,0)$ en el espacio del cuerpo. Si $R(t)$ especifica la rotación del cuerpo sobre el centro de masas, entonces un vector *r* del espacio del cuerpo será rotado al espacio del mundo $R(t)r$ en el tiempo *t.* Así, si p_0 es un punto arbitrario en el sólido rígido, en el espacio del cuerpo, entonces la rotación del espacio del mundo $p(t)$ de p_0 es el resultado de primero rotar p_0 sobre el origen y luego trasladarlo:

$$p(t) = R(t) + x(t)$$

Como el centro de masas esta en el origen, la localización en el espacio del mundo del centro de masas es dada directamente por $x(t)$. Esto permite dar un significado a $x(t)$ diciendo que $x(t)$ es la localización del centro de masas en el espacio del mundo en el tiempo t.

3.2.2. Velocidad lineal.

Es necesario saber cómo varían la posición y la orientación a lo largo del tiempo. Para ello necesitamos saber la derivada de $x(t)$ y de $R(t)$. Si $x(t)$ es la posición del centro de masas en el espacio del mundo, su derivada es la velocidad del centro de masas en el espacio del mundo:

$$v(t) = x'(t)$$

Si imaginamos que la orientación del cuerpo esta fija, entonces el único movimiento que el cuerpo puede hacer es una translación. $v(t)$ da la velocidad de este movimiento.

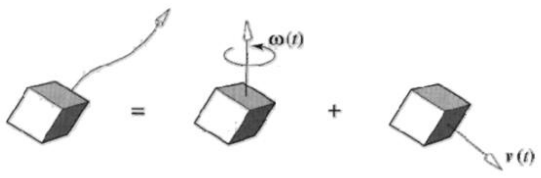


Figura 3.3: *Sólido rígido con velocidad angular y lineal.*

3.2.3. Velocidad angular.

Para representar un giro se utiliza el vector de velocidad angular w. Su modulo representa la rapidez de giro mientras que su dirección indica alrededor de que eje gira.

Mientras que $v(t)$ y $x(t)$ están relacionados por $v(t) = x'(t)$, para R(t) y w(t) no es tan simple ya que R´(t) no puede ser w(t) ya que R(t) es una matriz y w(t) es un vector. La expresión viene dada por:

$$\frac{dR}{dt} = \Omega R$$

Donde:

$$\Omega = \begin{pmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{pmatrix}$$

3.2.4. Masa del sólido rígido.

La masa de un sólido rígido es importante en el estudio de los comportamientos de estos cuerpos ya que influye en el cálculo de las velocidades tanto angular como lineal.

La masa se define como la resistencia que ofrece un cuerpo a ser desplazado. Aunque la masa de un cuerpo se puede calcular como la suma de todas las partículas elementales que lo componen, en nuestro motor estas se suponen dadas por el usuario.

El centro de masas de un sólido rígido es el punto alrededor del cual se encuentra la masa uniformemente distribuida. El centro de masas en el espacio del mundo se definiría:

$$CM = \frac{\sum m_i r_i(t)}{M}$$

Donde m_i es la masa de cada de las partículas que la componen, r_i es la posición actual de cada particula y M es la masa total del cuerpo. Tambien se obtiene que en las coordenadas del cuerpo:

$$CM = 0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

El momento de inercia indica la distribución radial de la masa de un cuerpo a lo largo de un eje. Este atributo indica la resistencia de un cuerpo a sufrir una rotación, pero dependiendo de qué eje se coja como referencia. Se puede expresar el momento de inercia como un tensor, que es un elemento matemático que tiene magnitud y dirección. El momento de inercia se puede formular como un tensor de segundo orden, en forma de matriz 3x3:

$$I = \begin{pmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{pmatrix}$$

Donde:

$$I_{xx} = \int (y^2 + z^2) dV \qquad I_{xy} = I_{yx} = \int xy \, dV$$

$$I_{yy} = \int (z^2 + x^2) dV \qquad I_{yz} = I_{zy} = \int yz \, dV$$

$$I_{zz} = \int (x^2 + y^2) dV \qquad I_{xz} = I_{zx} = \int zx \, dV$$

3.3. Springs.

Los Springs son uniones elásticas que simulan el comportamiento de resortes, y se rigen por la ley de elasticidad de **Robert Hooke**. En física, la ley de elasticidad de Hooke o ley de Hooke, originalmente formulada para casos del estiramiento longitudinal, establece que el alargamiento unitario que experimenta un material elástico es directamente proporcional a la fuerza aplicada F:

$$\in = \frac{\delta}{L} = \frac{F}{EA}$$

La forma más común de representar matemáticamente la Ley de Hooke es mediante la ecuación del muelle o resorte, donde se relaciona la fuerza F ejercida sobre el resorte con la elongación o alargamiento δ producido:

$$F = -K\delta$$

Donde *k* se llama constante elástica del resorte y δ es su elongación o variación que experimenta su longitud, en nuestro caso la longitud del vector que une sus extremos. Ésta fórmula es suficiente para resolver el problema pues aplicando ésta fuerza a los puntos de unión conseguimos simular la unión elástica entre cuerpos de una manera realista y precisa.

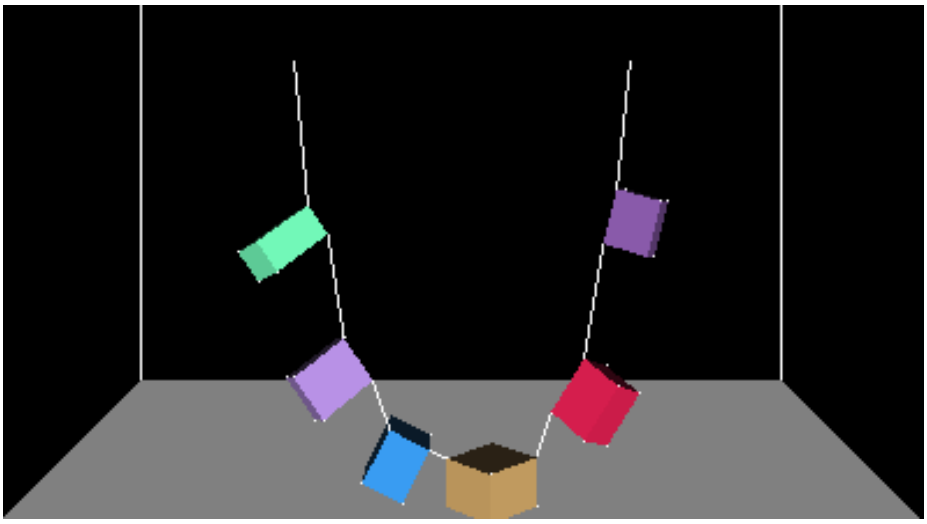


Figura 3.4: *Spring entre cuerpos.*

3.4. Detección de colisiones.

Para simular el comportamiento de las colisiones de los sólidos, el primer paso es determinar si existen colisiones en un estado del mundo determinado comparando los objetos dos a dos para seguidamente dar la respuesta adecuada en los casos positivos. Hay muchos métodos de detección de colisiones que se diferencian en complejidad y tiempo de cálculo, siendo unos adecuados solo para determinadas geometrías y otros genéricos que pueden ser utilizados con cualquier objeto.

La detección de colisiones es un problema muy costoso en tiempo de cálculo, ya que según el algoritmo utilizado puede llegar a necesitar comparar todos los vértices de un objeto con todos los vértices del otro, por éste motivo es frecuente establecer varios niveles de profundidad a la hora de detectar una colisión, de forma que primero se aplican los test más gruesos, que requieren menos tiempo de ejecución a modo de filtro, y luego se aplican los tests más precisos y que requieren una mayor cantidad de cálculos con los objetos que no fueron descartados en niveles anteriores.

Es importante destacar que los métodos de detección de colisiones utilizan solamente conceptos geométricos y no tienen en cuenta las características dinámicas del objeto como la velocidad o masa sino que solo tiene en cuenta su localización en el espacio y su geometría o forma.

3.4.1. Detección de colisiones esfera-esfera.

Ésta es la detección más sencilla de realizar y por tanto la que menos cálculos necesita. Basta con comprobar que la distancia entre los centros de las esferas es menor que la suma de sus radios para determinar que existe colisión entre ellas.

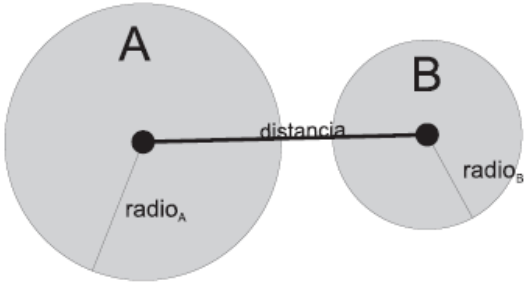


Figura 3.5: *Colisión esfera-esfera.*

Si se cumple que $|Xa - Xb| <= Ra + Rb$ entonces existe colision entre las esferas. El coste de ejecución de ésta comprobación tiene coste constante $\theta(1)$.

3.4.2. Detección de colisiones paralelepípedo-paralelepípedo.

Otro tipo de detección sencillo es la detección entre paralelepípedos alineados con el mundo, o detección de colisiones entre Bounding Boxes. Para la detección se comprueba si se solapan las proyecciones en el sistema de coordenadas del mundo, si todas las proyecciones se solapan estamos ante una colisión.

Figura 3.6: *Colisión paralelepípedo-paralelepípedo.*

Ésta detección es muy sencilla y eficiente pues el único cálculo que realice es comprobar si se solapan los intervalos dos a dos y es de coste constante.

3.4.3. Detección de colisiones vértice-plano.

Este tipo de colisión consiste en comprobar la distancia del punto(vértice) al plano en cuestión, dado que la normal de un plano da su orientación, si esta distancia es negativa , se puede decir que el punto esta "dentro" del plano. Es una comprobación cuyo coste es proporcional al número de planos y vértices que tengan los objetos sobre los que se realiza este test.

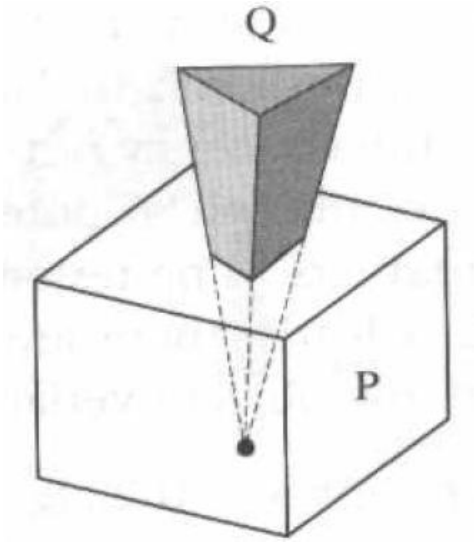


Figura 3.7: *Colisión vértice-esfera.*

Con este test, podemos comprobar que un vértice de un objeto está dentro de todos los planos del otro, lo que significa penetración

3.4.4. Detección de colisiones arista-plano.

En este caso, lo primero que se tiene que comprobar es que las rectas que contienen las aristas de un objeto, interseccionan con los planos de las caras del otro, entrando en juego más conceptos geométricos. Es una comprobación bastante costosa.

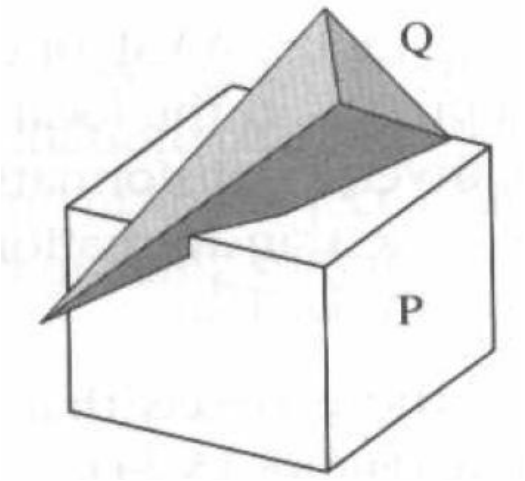


Figura 3.8: *Colisión vértice-esfera.*

Los puntos obtenidos de corte, se pasan al test de vértice-plano que es el encargado de detectar la colisión,

3.5. Respuesta a las colisiones.

Antes de definir el concepto de impulso que es al fin y al cabo el encargado de dar respuesta a las colisiones se debe introducir el concepto de velocidad relativa de un choque.

3.5.1. Velocidades relativas.

La velocidad relativa entre los objetos implicados en un choque, se define como el producto escalar de la normal del choque y la diferencia de las velocidades de los objetos (Velocidad cuerpo A – Velocidad cuerpo B). Tomando por convenio que en el punto de choque, la normal siempre va de B hacia A, en función del valor de la velocidad relativa se concluye:

- Vrel > 0: Los objetos se están alejando luego ya se ha dado respuesta a la colisión.
- Vrel = 0: Cuerpos en contacto estable, se procede a dormirlos. Este valor es idílico y se necesita un pequeño umbral de valores sobre el que poder decidir.
- Vrel < 0: Los cuerpos se están acercado luego la respuesta se dará solo en este caso.

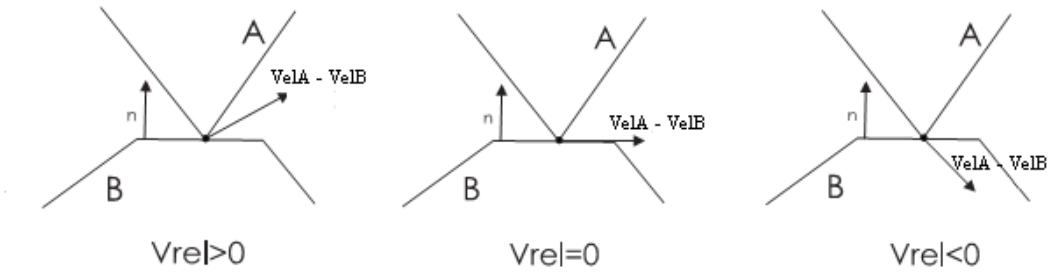


Figura 3.9: *Significado de la velocidad relativa en función de su valor*

3.5.2. Concepto de impulso.

El impulso **J,** se puede considerar como una gran fuerza que actúa durante un corto periodo de tiempo. Como explicamos antes, la velocidad tiene una componente lineal y otra angular, por cual vamos a explicar cómo se produce el cambio en cada componente.

El cambio en la velocidad lineal que produce J, si se aplica a un cuerpo de masa M es:

$$\Delta v = \frac{J}{M}$$

Esto equivale a que el cambio del momento lineal es ΔP = J. Al ser aplicado en un punto p, al igual que las fuerzas, produce un torque $\boldsymbol{\Gamma}$. Su valor es:

$$\Gamma impulso = (p - x(t))\ x\ J$$

$\Gamma impulso$ produce a su vez en cambio en el momento angular: ΔL = $\Gamma impulso$. Este cambio afecta a la velocidad angular en el momento t_0:

$$\Delta w = I^{-1}(t_0)\ \Gamma impulso$$

Cuando dos objetos colisionan, les aplicamos un impulso que cambia sus velocidades, por lo que podemos escribir el impulso con magnitud *j,* y con dirección de la normal de la colisión como:

$$J = j\boldsymbol{n}(t_0)$$

Por lo cual, el impulso actúa de manera positiva sobre el objeto A y de manera negativa sobre el objeto B

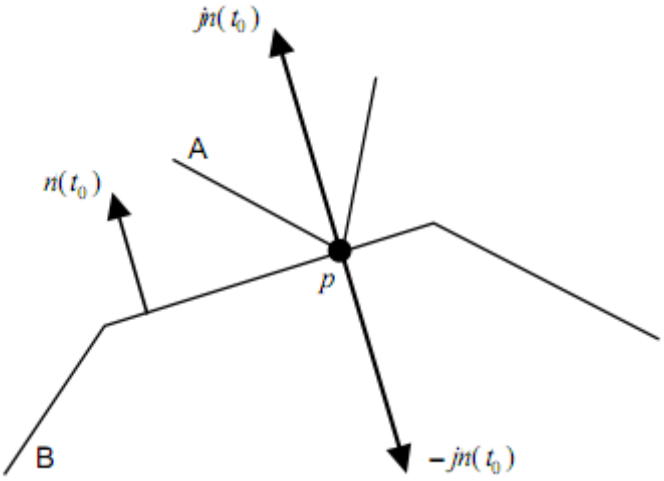


Figura 3.10: *Impulso.*

Según las leyes empíricas de colisión sin fricción se tiene que:

$$v_{rel}^{+} = \varepsilon v_{rel}^{-}$$

Siendo v_{rel}^{+} la velocidad relativa obtenida después de aplicar el impulso, v_{rel}^{-} la velocidad relativa de antes del impulso y ε el coeficiente de restitución, el cuál toma valores entre $0 \leq \varepsilon \leq 1$. Para calcular el impulso que tenemos que aplicar tendremos que calcular el valor de *j*, para ello expresaremos v_{rel}^{+} en función de v_{rel}^{-} y de *j*, para poder despejarlo con facilidad.

$$v_{relA}^{+} = v_{A}^{+} + w_{A}^{+} \; x \; r_{A}$$

Donde

$$r_{A} = p - x_{A}(t_0)$$

Por tanto

$$v_{A}^{+} = v_{A}^{-} + \Delta v = v_{A}^{-} + \frac{jn(t_0)}{M_A}$$

$$w_A^+ = w_A^- + \Delta w = w_A^- + I_A^{-1}(t_0)\ \Gamma impulso = w_A^- + I_A^{-1}\ \ (t_0)(r_A\ x\ jn(t_0))$$

Sustituyendo

$$v_{relA}^+ = v_A^- + \frac{jn\,(t_0)}{M_A} + (w_A^- + I_A^{-1}(t_0)(r_A\ x\ jn(t_0)))\ x\ r_A$$

$$= v_A^- + w_A^-\ x\ r_A + \frac{jn(t_0)}{M_A} + (I_A^{-1}(t_0)(r_A\ x\ jn(t_0)))\ x\ r_A$$

$$= v_{relA}^-\ + j(\frac{n(t_0)}{M_A} + I_A^{-1}(t_0)\big(r_A\ x\ n(t_0)\big)\ x\ r_A)$$

Realizamos los mismos cálculos para la velocidad del punto de B, pero teniendo en cuenta que el impulso tiene el signo contrario:

$$v_{relB}^+ = v_{relB}^-\ -\ j(\frac{n(t_0)}{M_B} + I_B^{-1}(t_0)\big(r_B\ x\ n(t_0)\big)\ x\ r_B)$$

Calculamos la resta de las velocidades relativas

$$v_{relA}^+ - v_{relB}^+ = (v_{relA}^- - v_{relB}^-) + j(\frac{n(t_0)}{M_A} + \frac{n(t_0)}{M_B} + (I_A^{-1}(t_0)(r_A\ x\ n(t_0)))\ x\ r_A + (I_B^{-1}(t_0)(r_B\ x\ n(t_0)))\ x\ r_B)$$

Calculamos v_{rel}^+, teniendo en cuenta que $n(t_0)$ tiene módulo 1, con lo cuál $n(t_0).n(t_0) = 1$

$$v_{rel}^+ = n(t_0)(v_{relA}^+ - v_{relB}^+)$$

$$v_{rel}^{+} = n(t_0)((v_{relA}^{-} - v_{relB}^{-}) + j\left(\frac{n(t_0)}{M_A} + \frac{n(t_0)}{M_B} + (I_A^{-1}(t_0)(r_A \; x \; n(t_0))) \; x \; r_A + (I_B^{-1}(t_0)(r_B \; x \; n(t_0))) \; x \; r_B\right)$$

$$v_{rel}^{+} = v_{rel}^{-} + j\left(\frac{1}{M_A} + \frac{1}{M_B} + (I_A^{-1}(t_0)(r_A \; x \; n(t_0))) \; x \; r_A + (I_B^{-1}(t_0)(r_B \; x \; n(t_0))) \; x \; r_B\right)$$

Sabiendo que $v_{rel}^{+} = \varepsilon v_{rel}^{-}$ y sustituyendo obtenemos la magnitud del impulso:

$$j = \frac{-(1+\varepsilon)v_{rel}^{-}}{(1/M_A + 1/M_B + \left(I_A^{-1}(t_0)\left(r_A \; x \; n(t_0)\right)\right) x \; r_A + \left(I_B^{-1}(t_0)\left(r_B \; x \; n(t_0)\right)\right) x \; r_B)}$$

4. Resolución de Ecuaciones Diferenciales Ordinarias.

Los motores de física requieren la resolución de ecuaciones diferenciales, ya que necesitan saber los valores de magnitudes derivadas como la velocidad y aceleración.

Hay una gran variedad de resolutores numéricos para las ecuaciones diferenciales como pueden ser el método del punto medio, método de Runge-Kutta 4 o el método de Euler, siendo este último el que se implemento por su sencillez.

4.1. Problemas de valor inicial.

Las ecuaciones diferenciales describen la relación entre una función desconocida y sus derivadas. Resolver una ecuación diferencial es encontrar una función que satisface esta relación, cumpliendo algunas condiciones adicionales. Este motor se ocupa principalmente de un tipo particular de problemas, llamado *problemas de valor inicial*. En un problema de valor inicial canónico, el comportamiento del sistema se describe mediante una ecuación diferencial ordinaria (EDO) de la forma

$$x' = f(x, t)$$

En dos dimensiones $x(t)$ dibuja una curva que describe el movimiento de un punto en el plano. En cualquier punto x de la función f puede ser evaluado para proporcionar un vector de dos dimensiones, por lo que f define un campo de vectores en el plano (ver figura 1). El vector en x es la velocidad que el punto móvil **p** en movimiento debe tener si es que se mueve a través de x (si se mueve, aunque podría permanecer inmóvil). El futuro movimiento estará definido por f. La trayectoria descrita por p a través de f forma la curva integral del campo vectorial.

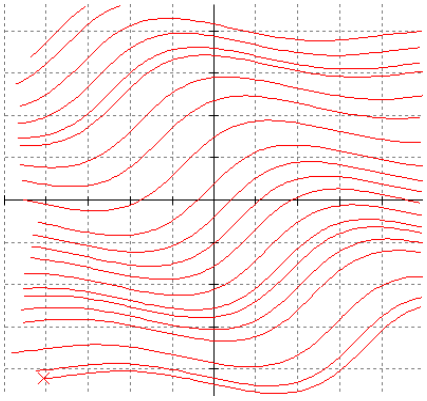


***Figura 4. 1**. Curvas solución a un problema de valor inicial.*

Se describió f como una función tanto de x y t, pero la función derivada puede o no puede depender directamente del tiempo. Si lo hace, entonces no sólo el punto $\boldsymbol{p}$ se mueve, el campo vectorial del mismo se mueve, de modo que la velocidad no sólo depende de dónde está, sino también de cuando llegue allí. En ese caso, la derivada x' depende en el tiempo de dos formas: en primer lugar, los vectores derivados se mueven, y en segundo lugar, el punto p, porque se mueve en una trayectoria $x(t)$, ve diferentes vectores derivados en diferentes momentos. Esta dualidad en el tiempo no debe llevar a confusión si se toma como referencia el dibujo de una partícula flotando a lo largo de un campo vectorial ondulado.

4.2. Soluciones numéricas.

El motor se centra exclusivamente en la resolución numérica, en la cual se toman pasos discretos tomando como inicio $x(t_0)$. Para tomar un paso se utiliza la función derivada f para calcular un cambio aproximado en x , Δx, a lo largo de un intervalo de tiempo Δt, para luego incrementar x en Δx para obtener un nuevo valor de x. Para calcular la solución numérica, la función derivada f es considerada como una caja negra: se proporcionan valores numéricos para x y t, recibiendo a cambio un valor numérico para x'. Los métodos numéricos operan haciendo una o más de estas evaluaciones en cada paso.

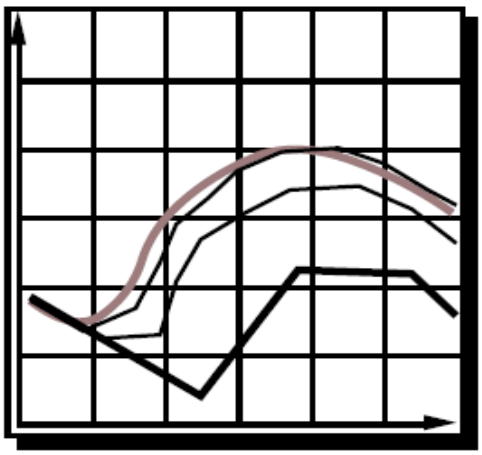
4.2.1. Método de Euler.

El método numérico más simple se llama el método de Euler. Llamaremos a nuestro valor inicial para x , como $x_0 = x(t_0)$ y una estimación de x a un instante

posterior $t_0 + h$ como $x(t_0 + h)$ donde h es un parámetro de paso. El método de Euler simplemente calcula tomando un paso en la derivación,

$$x(t_0 + h) = x_0 + hx'(t_0)$$

Se puede utilizar un campo vectorial de dos dimensiones para visualizar el método de Euler. En lugar de la curva integral real, p se sigue un camino poligonal, cada lado del cual se determina mediante la evaluación del vector f al principio e incrementándolo por$h$. Véase la figura 4. 2.



-Pasos de tiempo discretos.

-A mayor tamaño de paso mayor error.

$$x(t + \Delta t) = x(t) + \Delta t\, f(x, t)$$

Figura 4.2 .*En vez de la curva integral real, la solución aproximada sigue un camino poligonal, obtenido al evaluarse la derivada al inicio de cada rama. Aquí se muestra como la precisión de la solución empeora con el incremento del paso.*

Aunque simple, el método de Euler no es exacto. Consideremos el caso de una función de dos dimensiones f cuyas curvas integrales son círculos concéntricos. Un punto p regido por f se supone que siempre tiene que orbitar en el mismo circulo que empezó. En cambio, con cada paso de Euler, p se moverá en línea recta a un círculo de mayor radio, de modo que su trayectoria siga una espiral hacia afuera. La disminución de la amplitud de paso se reduce la tasa de esta deriva hacia el exterior, pero nunca la eliminara completamente.

Por otra parte, el método de Euler puede ser inestable. Se considera una función de una dimensión $f = -kx$, que debería disminuir exponencialmente el punto p a cero. Para amplitudes de paso suficientemente pequeñas se obtiene un comportamiento razonable, pero cuando $h > 1/k$, tenemos que $|\Delta x| > |x|$, por lo que

la solución oscila alrededor de cero. Más allá de la oscilación diverge y el sistema se colapsa. Véase la figura 4.

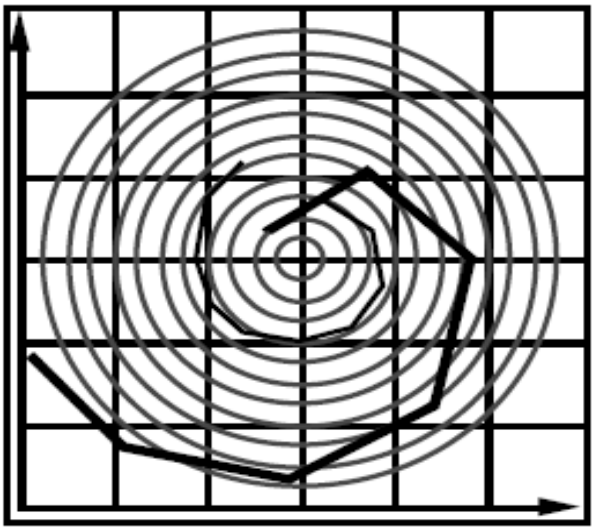


Figura 4.3 *Poca precisión: El error convierte a x(t) en una espiral de nuestra elección en vez de en un círculo.*

El método de Euler no es eficaz. La mayoría de los métodos de solución numérica pasan casi todos su tiempo de realizando las evaluaciones derivadas, por lo que el coste computacional *por paso* está determinado por el número de evaluaciones por paso. Aunque el método de Euler sólo requiere de una evaluación cada paso, la verdadera eficacia de un método depende del tamaño del tamaño del paso que este le permite dar(preservando precisión y estabilidad) así como el coste por paso.

4.2.2. Método del punto medio.

El método del punto medio es un método de segundo orden del algoritmo Runge-Kutta. El valor inicial es $(t_0, \vec{X}_0)$.Las aproximaciones sucesivas $(t_i, \vec{X}_i)$.para $i > 0$ son generadas por:

$$\vec{A}_1 = \vec{F}(t_i, \vec{X}_i)$$

$$\vec{A}_2 = \vec{F}(t_i + \frac{h}{2}, \vec{X}_i + h\,\vec{A}_1/2)$$

$$\vec{X}_{i+1} = \vec{X}_{\,i} + h\vec{A}_2)$$

$$t_{(i+1)} = t_i + h$$

4.2.3. Runge-Kutta de cuarto orden.

El método de Runge-Kutta de cuarto orden tienes una precisión elevada. El valor inicial es $(t_0, \vec{X}_0)$. Sucesivas aproximaciones $(t_i, \vec{X}_i)$ para $i > 0$ están generadas por:

$$\vec{A}_1 = \vec{F}(t_i, \vec{X}_i)$$

$$\vec{A}_2 = \vec{F}(t_i + \frac{h}{2}, \vec{X}_i + h\,\vec{A}_1/2)$$

$$\vec{A}_3 = \vec{F}(t_i + \frac{h}{2}, \vec{X}_i + h\,\vec{A}_2/2)$$

$$\vec{A}_4 = \vec{F}(t_i + h, \vec{X}_i + h\,\vec{A}_3)$$

$$\vec{X}_{i+1} = \vec{X}_{\,i} + \frac{h}{6}(\vec{A}_1 + 2\vec{A}_2 + 2\vec{A}_3 + \vec{A}_4)$$

$$t_{(i+1)} = t_i + h$$

5. Arquitectura del motor.

En este apartado para explicar la arquitectura del motor se muestra un diagrama de clases que permite ver la dependencia entre estas. Posteriormente se explican los diferentes módulos por separado.

5.1. Diagrama de clases.

El diagrama que se muestra a continuación muestra, las principales clases que forman nuestro motor y como se unen entre ellas. Se han obviado algunas clases que sirven de apoyo al proyecto, como las que implementan las operaciones matemáticas o la plantilla de las listas de objetos.

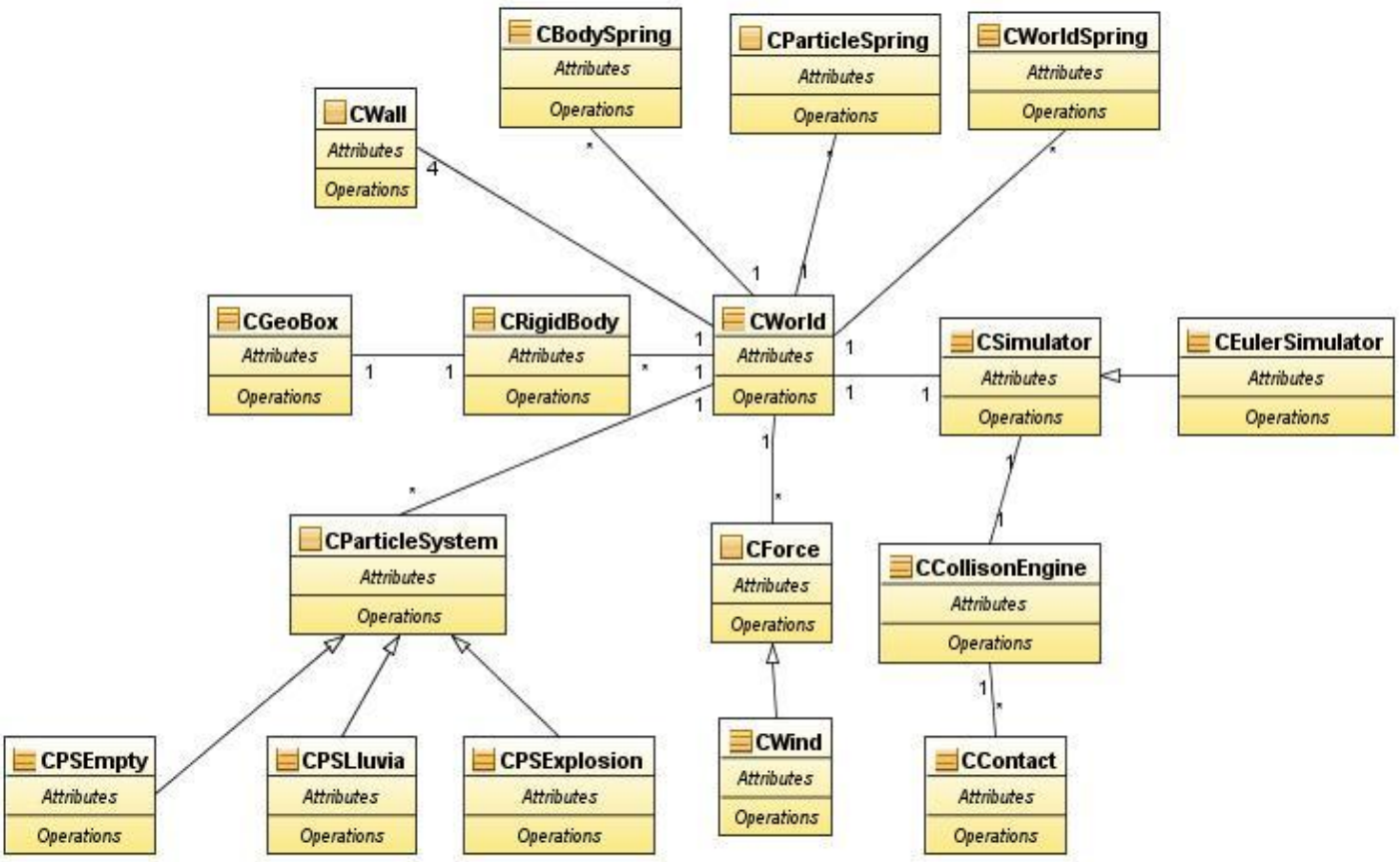


Figura 5.1: *Diagrama de clases de la aplicación*

Para poder llevar a cabo simulaciones, necesitamos tener definido el estado del mundo sobre el que operaremos para llegar al estado siguiente tras transcurrir un tiempo determinado, la clase CWorld es la que define el estado, y la clase CSimulator es la encargada de operar sobre el estado actual apoyándose de la clase CCollisionEngine para la detección y respuesta de colisiones.

La clase CWorld contendrá todos los objetos que componen el mundo a simular, estos objetos pueden ser del tipo:

- CWall: Muros o paredes que delimitan el mundo.
- CRigidBody: Sólidos rígidos.
- CWorldSpring: Uniones elásticas entre sólidos y el mundo.
- CBodySpring: Uniones elásticas entre dos sólidos.
- CParticleSystem: Sistemas de partículas.
- CForce: Fuerzas que actúan sobre los objetos.

La clase CSimulator es la encargada de operar con el estado del mundo, hay diferentes métodos de simulación que extienden esta clase, en nuestro caso hemos utilizado el método de integración de Euler con la ayuda de la clase CEulerSimulator, pero el sistema permite añadir distintos métodos a utilizar que extiendan de la clase CSimulator.

Por último tenemos el módulo de detección y respuesta a las colisiones que denominamos CCollisionEngine y que contiene los métodos que posibilita la detección de colisiones entre distintos objetos del mundo y da la respuesta adecuada en caso de detección. Ésta clase es utilizada por la clase CSimulator para la simulación.

5.2. Mundo.

5.2.1. Clase CWorld.

La clase CWorld representa el mundo que se va a simular, es un contenedor para todos los objetos y características del mismo y además ofrece métodos para la modificación y consulta del estado.

Un mundo se define por: El tamaño para sus tres coordenadas (ancho, alto, largo), el vector de gravedad y las listas de objetos que contiene (muros, rígidos, springs, particlesystems y fuerzas).

Para crear un mundo basta con llamar a su constructor pasando como parámetros las dimensiones que deseemos, esto creará un mundo vacío de las dimensiones indicadas con un vector de gravedad nulo.

```
CWorld * pWorld = new CWorld(LongX, LongY, LongZ);
```

Para obtener el valor de las dimensiones podemos utilizar los siguientes métodos:

```
real WorldXLength = pWorld->GetWorldXLength();
real WorldYLength = pWorld->GetWorldYLength();
real WorldZLength = pWorld->GetWorldZLength();
```

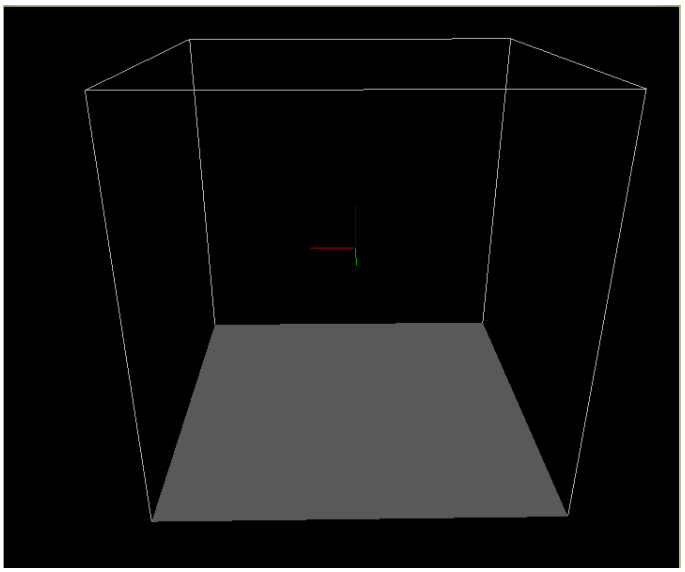


Figura 5.2:*Representación del mundo*

En la mayoría de los casos querremos incluir una aceleración en alguna dirección que afecte a todos los objetos por igual, esto lo haremos cambiando el GravityVector a través del siguiente método que recibe como parámetro el vector deseado:

```
pWorld->SetGravityVector(vector_3(r(0),r(0),r(-10)));
```

Para obtener el valor del GravityVector utilizaremos:

```
vector_3 gravity = pWorld->GetGravityVector();
```

Se pueden añadir planos para delimitar el mundo, estos planos actúan como paredes, conteniendo a los cuerpos dentro del espacio que delimitan, para añadirlos se utiliza el siguiente método que recibe como parámetros la instancia del objeto CWall que queremos añadir.

```
pWorld->AddWall(wall);
```

Para averiguar el número de muros que hay en el mundo:

```
unsigned int numMuros = pWorld->GetNumWalls();
```

Para eliminar el muro en la posición 'i':

```
pWorld->RemoveWallPos(i);
```

Para eliminar un muro determinado (CWall* w):

```
pWorld->RemoveWall(w);
```

Podemos acceder al muro en la posición 'i' mediante

```
CWall* wall = pWorld->GetWall(unsigned int i);
```

Para agregar cuerpos rígidos al mundo haremos uso del siguiente método que recibe como parámetro la instancia de la clase CRigidBody que queremos añadir:

```
pWorld->AddBody(Body);
```

Para averiguar el numero de sólidos que hay en el mundo

```
unsigned int numBodies = pWorld->GetNumBodies();
```

Para eliminar el sólido en la posición 'i':

```
pWorld->RemoveBody(i);
```

Para eliminar un sólido determinado (CRigidBody* w):

```
pWorld->RemoveBody(w);
```

Podemos acceder al sólido en la posición ‘i’ mediante

```
CRigidBody* body = pWorld->GetBody(unsigned int i);
```

Los WorldSprings son uniones elásticas entre un punto fijo del espacio y un vértice de un determinado cuerpo, para añadirlas o eliminarlas se utiliza el siguiente método que recibe como parámetro la instancia de la clase CWorldSpring que queremos añadir o eliminar.

```
pWorld->AddWorldSpring(worldSpring);
```

Para averiguar el numero de WorldSprings que hay en el mundo

```
unsigned int numWSprings = pWorld->GetNumWorldSprings();
```

Para eliminar el WorldSpring en la posición ‘i’:

```
pWorld->RemoveWorldSpring(i);
```

Para eliminar un WorldSpring determinado (CWorldSpring * w):

```
pWorld->RemoveWorldSpring(w);
```

Podemos acceder al WorldSpring en la posición 'i' mediante

```
CWorldSpring* ws = pWorld->GetWorldSpring(unsigned int i);
```

Los BodySprings son uniones elásticas entre dos sólido. Los métodos son similares a los que tratan con los WorldSpring:

```
pWorld->AddBodySpring(bodySpring);
unsigned int numBSprings = pWorld->GetNumBodySprings();
pWorld->RemoveBodySpring(i);
pWorld->RemoveBodySpring(b);
CBodySpring* bs = pWorld->GetBodySpring(unsigned int i);
```

Los ParticleSprings son uniones elásticas entre dos partículas o una partícula y un punto del espacio. Sus métodos son:

```
pWorld->AddParticleSpring(particleSpring);
unsigned int numPSprings = pWorld->GetNumParticleSprings();
```

```
pWorld->RemoveParticleSpring(i);
pWorld->RemoveParticleSpring(b);
CParticlepring* bs = pWorld->GetParticleSpring(unsigned int i);
```

Para agregar/eliminar Sistemas de partículas haremos uso de los siguientes métodos que reciben como parámetro la instancia de la clase CParticleSystem que queremos añadir o eliminar:

```
pWorld->AddParticleSystem(ps);
```

Para averiguar el numero de ParticleSystems que hay en el mundo

```
unsigned int numPs = pWorld->GetNumParticleSystems();
```

Para eliminar el ParticleSystem en la posición 'i':

```
pWorld->RemoveParticleSystem (i);
```

Para eliminar un ParticleSystemdeterminado (CParticleSystem* ps):

```
pWorld->RemoveParticleSystem (ps);
```

Podemos acceder al ParticleSystem en la posición 'i' mediante

```
CParticleSystem* ps = pWorld->GetParticleSystem(unsigned int i);
```

5.3. Sistemas de partículas.

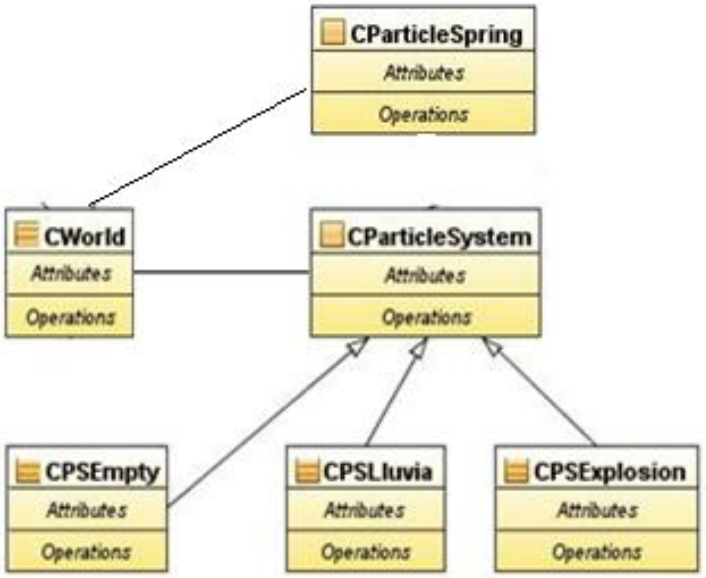


Figura 5.3: *Clases correspondientes a partículas*

5.3.1. Clase CParticleSystem (clase abstracta).

Clase contenedor de partículas, clase abstracta que ofrece los métodos básicos para el manejo de sistemas de partículas. Sus atributos son los siguientes:

```
bool collisionCheckActive; /* Indica si se resuelven las colisiones para este sistema*/
tLista<CParticle*> *aParticles; /* Array de punteros a partículas */
real Kvd; /* Viscous Drag constant */
real particlesMass; /* masa aproximada de las partículas*/
real varMass; /* masa de las partículas*/
```

```
int numMaxParticles;  /* nº maximo de partículas*/
eeal color[4];  /* Color de las particulas*/
real varColor[4];   /* Variación del color de las particulas*/
real life;  /* Vida de las particulas, -1 = no mueren*/
real varLife;  /* Variación de la vida */
real tam;  /* Tamaño de las particulas*/
real varTam;  /* Variación del tamaño de las particulas*/
```

Los métodos que ofrece son:

- Constructor:

```
CParticleSystem(real mass,real varMass, real Kvd, int
numParticles,real color[4],real varColor[4],real life,real
varLife,real tam,real varTam);
```

- Destructor:

```
virtual ~CParticleSystem();
```

- Acceso y modificación de atributos:

```
int GetNumParticles();
real GetKvd();
void SetCollisionCheckActive(bool v);
inline bool GetCollisionCheckActive();
CParticle* GetParticle(int i);
void RemoveParticle(int i);
real GetLife();
```

Métodos virtuales puros que deben implementar los distintos sistemas de partículas que se creen.

Para controlar la acción a realizar en caso de una colisión se implementara el siguiente método pasando como parámetro la partícula que colisiona y su posición en la lista de partículas.

```
virtual void ParticleCollisionEvent(CParticle*p,int pos)=0;
```

Para controlar la acción a realizar para cada paso de simulación se utiliza el siguiente método pasándole como parámetros la partícula a simular, su posición en la lista y el tiempo que hay que simular.

```
virtual bool SimulateStepParticle(CParticle*p,int pos,real
DeltaTime)=0;
```

5.3.1.1. Clase CParticle.

Clase interna a CParticleSystem, que implementa el comportamiento de las partículas, sus atributos son:

```
real  m; /* Masa */
vector_3  x; /* Posición */
vector_3  v;/* Velocidad */
vector_3  f;/* Acumulador de fuerzas */
real l;/* Tiempo de vida restante*/
real tam;/* Tamaño*/
real color[4];/* Color*/
```

Los métodos que ofrece son:

- Constructor:

```
CParticle(real m, vector_3 x,vector_3 v,real l,real tam,
real color[4]);
```

- Acceso y modificación de atributos:

```
vector_3 GetPos();
vector_3 GetVel();
void SetVel(vector_3 v);
vector_3 GetForce();
void SetForce(vector_3 f);
void AddForce(vector_3 f);
real GetLife();
real GetTam();
real GetColor();
real GetMass();
```

5.3.2. Clase CPSExplosion.

Sistema de partículas con las siguientes características adicionales:

- Origen: punto en el que nacen todas las partículas.
- Velocidad mínima de las partículas.
- Variación de la velocidad de cada partícula.

A partir de éstos valores se crean las partículas con una velocidad de dirección aleatoria y módulo la velocidad mínima más una variación aleatoria de forma que simulan explosiones ya que cada partícula es lanzada a una velocidad y dirección semi aleatorias.

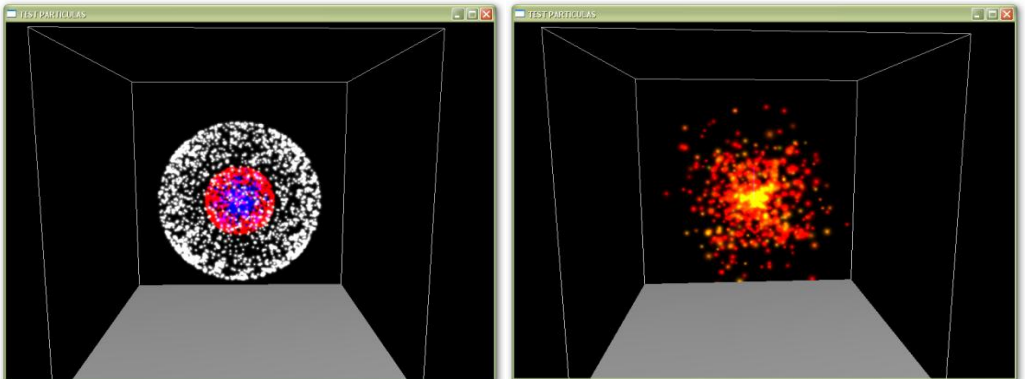


Figura 5.4: *Ejemplos de sistemas de partículas*

5.3.3. Clase CPSLluvia.

Sistema de partículas con las siguientes características adicionales:

- Origen: localización del sistema en el mundo.
- Márgenes: plano sobre el que nacen las partículas.
- Altura muerte: representa el suelo, punto en el que la partícula se elimina para volver a crearse
- Velocidad inicial: módulo y dirección de la velocidad de las partículas.
- Variación de la velocidad: vector que indica cómo puede variar la velocidad de cada partícula.

Con éstos valores se parametrizan éstos sistemas de partículas, creados para dar soporte a efectos del tipo humo, lluvia, nieve, chorros de partículas.

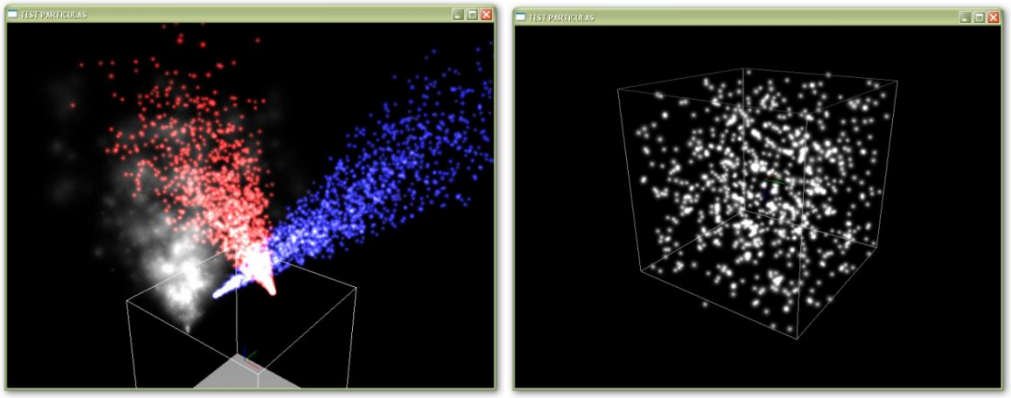


Figura 5.5: *Mas **e**jemplos de sistemas de partículas*

5.3.4. Clase CPSEmpty.

Sistema de partículas neutro o vacio, es simplemente un contenedor de partículas, utilizado para trabajar manualmente con ellas. A este contenedor se le pueden añadir partículas para crear objetos complejos como sólidos deformables y telas.

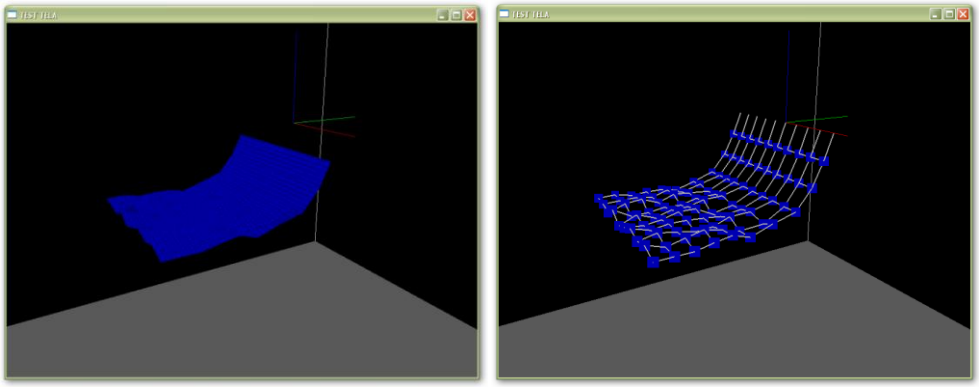


Figura 5.6: *Ejemplo de aplicación de CPSEmpty*

5.3.5. Clase CParticleSpring.

Un ParticleSpring es una unión elástica entre dos partículas. Hay dos constructores para crear los dos tipos de ParticleSprings que existen.

ParticleSpring entre partículas: Se llama al constructor pasándole como parámetros las dos partículas, la constante de Hooke que lo caracteriza , una constante de amortiguación y la longitud que tiene el Spring.

```
CParticleSpring(CParticle *p1,CParticle *p2,real Ks,real Kd,real
longitud);
```

ParticleSpring entre partícula y punto del mundo: Se llama al constructor pasándole como parámetros la partícula, el punto del mundo al que se engancha, la

constante de Hooke que lo caracteriza , una constante de amortiguación y la longitud que tiene el Spring.

5.4. Sólidos rígidos

Figura 5.7: *Clases de Sólidos Rígidos*

5.4.1. Clase CRigidBody.

Para la simulación de sólidos rígidos se ha creado a clase CRigidBody, ésta clase tiene todos los datos necesarios para definir un sólido y se apoya en una lista de configuraciones (CConfig), cada configuración se almacenan los datos del sólido que varían en cada simulación de forma que se tienen varias configuraciones almacenadas que deben de ser gestionadas por el resolutor. Por ejemplo, el resolutor de Euler solo necesita dos configuraciones, anterior y presente. También tienen un objeto de la clase CGeoBox que define su geometría para ser utilizada por el motor de colisiones.

```
real Kdl;          // factor de amortiguamiento lineal
real Kda;          // factor de amortiguamiento angular
real OneOverMass; //inversa de la masa
matrix_3x3  InverseBodyInertiaTensor;//inversa  del  tensor  de
inercia
real CoefficientOfRestitution;//coeficiente de rozamiento
tLista<vector_3> *aBodyBoundingVertices;  //Array de vértices que
componen el objeto tomando el origen como referencia
CGeoBox*  boundingBox;//Estructura  auxiliar  que  define  la
geometría del objeto
bool inmovil,dormido;//valores booleanos de estado.
```

```
CConfig ** aConfigurations;//array de configuraciones.
```

Los métodos que ofrece la clase CRigidBody son:

- Constructor: Crea un sólido con una geometría determinada:

```
CRigidBody(CGeoBox* box);
```

- Destructor: Para destruir un sólido:

```
 ~CRigidBody();
```

- CalculateVertices: Método que calcula los vértices de la configuración indicada según su matriz de orientación y el centro de masas.

```
void CalculateVertices(int ConfigurationIndex);
```

- Acceso y modificación de sus atributos:

```
real GetKdl(void){return Kdl;};
void SetKdl(real v){Kdl=v;};
real GetKda(void){return Kda;};
void SetKda(real v){Kda=v;};
bool GetInmovil(void);
bool GetDormido(void);
void SetInmovil(bool v);
void SetDormido(bool v);
real GetOneOverMass(void);
void SetOneOverMass(real v);
```

```
real GetCoefficientOfRestitution(void);
void SetCoefficientOfRestitution(real v);
matrix_3x3 GetInverseBodyInertiaTensor(void);
void SetInverseBodyInertiaTensor(matrix_3x3 m);
CConfig* GetConfiguration(int i);
void Integrate(real DeltaTime,int ConfIndex,int Target);
int GetNumberOfBoundingVertices(void);
vector_3 GetBodyBoundingVertice(int i);
void AddBodyBoundingVertice(vector_3 v);
CGeoBox* GetGeoBox();
```

5.4.1.1 .Clase CConfig,

La clase CConfig, es una clase interna de CRigidBody, contiene los datos de un sólido que pueden variar durante un paso de simulación estos atributos son:

```
vector_3 CMVelocity; //velocidad del centro de masas.
vector_3 AngularMomentum; //momento angular del objeto.
vector_3 CMPosition; //posición del centro de masas.
matrix_3x3 Orientation; //orientación del objeto.
matrix_3x3 InverseWorldInertiaTensor; //inversa del tensor de inercia
vector_3 AngularVelocity; //velocidad angular
vector_3 CMForce; //acumulador de fuerzas
vector_3 Torque; //acumulador de fuerzas de rotación
tLista<vector_3> *aBoundingVertices; //lista de vértices tomando como referencia el centro de masas.
```

Los métodos que ofrece la clase CConfig son todos los necesarios para permitir el acceso y modificación de todos sus atributos.

5.4.2. Clase CGeoBox.

Esta clase contiene la información de la geometría del objeto, la utiliza principalmente el motor de colisiones. Sus atributos son:

```
float radio; //para una primera aproximación
tLista<CGeoBox::QuadFace>* lCaras;  //lista de caras del objeto
```

Los métodos que ofrece la clase CGeoBox son todos los necesarios para permitir el acceso y modificación de todos sus atributos.

```
CGeoBox(float radio);
~CGeoBox();
void AddCara(QuadFace lv);
QuadFace GetCara(int index);
float GetRadio();
int GetNumCaras();
int GetNumAristas();
vector_3* GetArista(int i,tLista<vector_3>* lista);
```

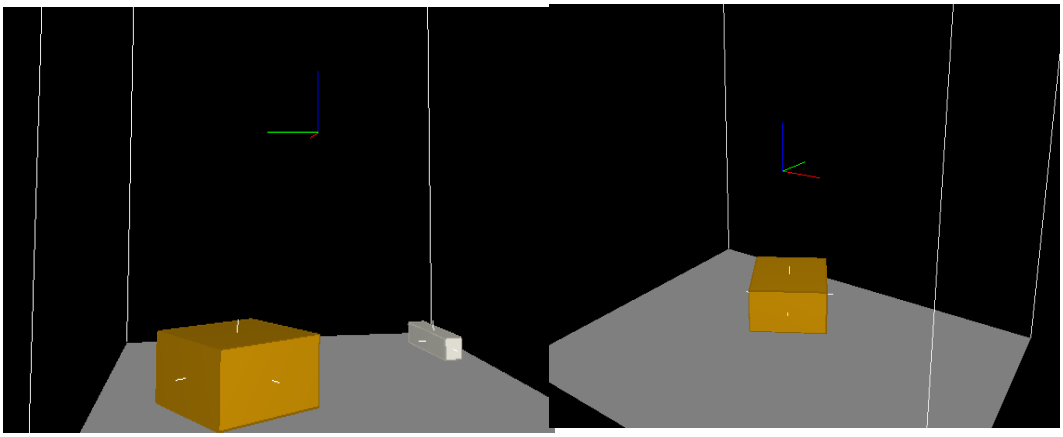


Figura 5.8 y 5.9: *Ejemplos de sólidos rígidos*

5.5. Módulo de simulación.

Figura 5.10: *Clases para la simulación*

5.5.1. Clase CSimulator (abstracta, permite distintos simuladores).

Es la encargada de simular el comportamiento de los objetos que hay dentro del mundo que gestiona. Tiene los siguientes métodos.

- Constructor: Necesita como parámetro un puntero al mundo con el que va a trabajar.

```
CSimulator( CWorld* pWorld);
```

- Simulate: Este método avanzará la simulación DeltaTime segundos, cada tipo de simulador deberá implementar la suya.

```
virtual void Simulate( real DeltaTime );
```

- SetMaxTimeStep: Para cambiar el tamaño del paso de simulación:

```
void SetMaxTimeStep(float d);
```

- Para cambiar el DephEpsilon en objetos y en partículas:

```
void SetDepthEpsilon(float d);
void SetParticleDepthEpsilon(float d);
```

5.5.2. Clase CEulerSimulator (extiende CSimulator) .

Es la encargada de simular el comportamiento de los objetos que hay dentro del mundo que gestiona utilizando el método de integración de Euler para simular. Extiende a la clase CSimulator e implementa los métodos explicados anteriormente., mediante Euler.

- Constructor: Necesita como parámetro un puntero al mundo con el que va a trabajar.

```
CEulerSimulator( CWorld* pWorld);
```

5.6. Muros.

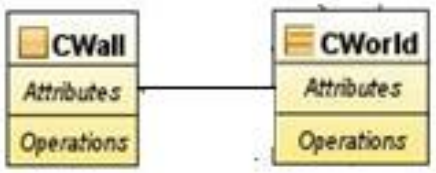


Figura 5.11: *Clases para los muros*

5.6.1. Clase CWall.

Para crear muros debemos invocar el método constructor que recibe como parámetros el vector_3(a,b,c) y el real d que corresponden con la ecuación “ax + by + cz + d = 0” del muro que queremos crear.

```
CWall muro = CWall(vector_3 Normal,float d);
```

Para consultar los atributos haremos uso de sus métodos accesores:

```
vector_3 GetNormal();
float GetD();
```

5.7. Springs.

Figura 5.12: *Clases para Springs*

5.7.1. Clase CWorldSpring.

Un WorldSpring es una unión elástica entre un punto del espacio y un vértice de un cuerpo. Para crear un WorldSpring llamaremos a su constructor pasándole como parámetros: el índice que identifica el cuerpo, número de vértice con el que se engancha el primer extremo, punto del espacio al que se engancha el otro extremo, la constante de Hooke que lo caracteriza y por último una constante de amortiguación.

```
CWorldSpring(int unsigned B, int unsigned V, vector_3 const &A ,
real Ks, real Kd ) ;
```

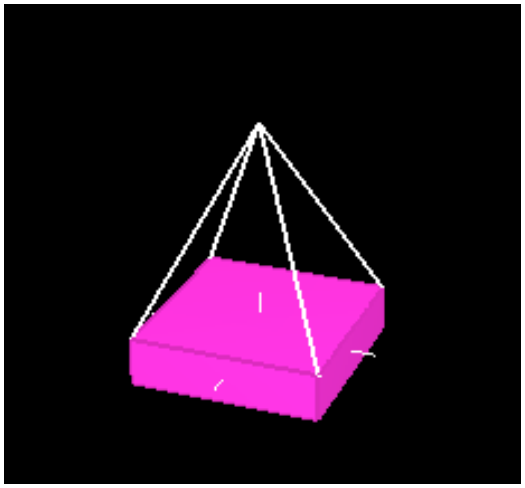


Figura 5.13: *Ejemplo de sólido rígido con WorldSprings*

5.7.2. Clase CBodySpring.

Un BodySpring es una unión elástica entre dos vértices. Para crear un BodySpring llamaremos a su constructor pasándole como parámetros: el índice que identifica al primer cuerpo, numero de vértice con el que se engancha el primer extremo, el índice que identifica al segundo cuerpo, número de vértice al que se engancha el otro extremo, la constante de Hooke que lo caracteriza y por último una constante de amortiguación.

```
CBodySpring(int unsigned B0,int unsigned V0,int unsigned B1,int
V1, real Ks, real Kd);
```

5.8. Fuerzas.

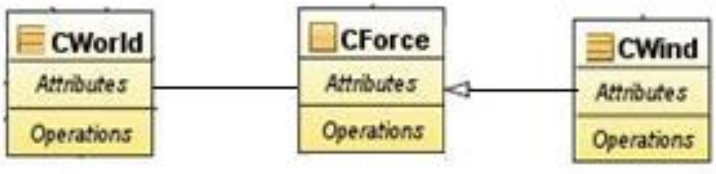


Figura 5.14: *Clases para fuerzas*

5.8.1. Clase CForce (clase abstracta).

Es la encargada de simular el comportamiento de las fuerzas, es una clase abstracta que ofrece los métodos que se deben redefinir. Tiene los siguientes métodos.

- Constructor: Necesita como parámetro una dirección para fuerza que se quiera aplicar

```
CForce(vector_3 direction);
```

- SimulateStepForce: Este método avanzará la simulación DeltaTime segundos, cada tipo de fuerza implementada se comportará según su actúe su física.

```
virtual void SimulateStepForce(real DeltaTime)=0;
```

- GetForce: Devuelve la fuerza.

```
virtual vector_3 GetForce()=0;
```

- GetDirectionVector: Devuelve la dirección de la fuerza

```
vector_3 GetDirectionVector();
```

5.8.2. Clase CWind(Extiende CForce).

Simula el comportamiento de un viento racheado mediante una función sinusoidal compuesta. Extiende la clase CForce

- Constructor: Necesita como parámetro una dirección para fuerza que se quiera aplicar, una magnitud y un tiempo de racha

```
CFWind(vector_3 direction,real initalMag,real run);
```

- SimulateStepForce: Este método avanzará la simulación DeltaTime segundos, simulando el viento

```
void SimulateStepForce(real DeltaTime);
```

- GetForce: Devuelve la fuerza.

```
vector_3 GetForce();
```

6. Implementación del motor de colisiones.

6.1. Detección de colisiones.

Cuando se han integrado las fuerzas y calculadas las nuevas posiciones de los cuerpos, entra en juego la detección de colisiones. Este módulo es una de las partes más importantes del proyecto ya que buena parte del funcionamiento correcto de las simulaciones dependen de su comportamiento. A la hora de detectar las colisiones y calcular los correspondientes puntos de contacto, realizaremos varias aproximaciones, realizando una primera aproximación a nivel grueso, para luego pasar a refinarla con la aproximación a nivel fino.

Cabe destacar la presencia de la constante DepthEpsilon, la cual debido a errores de redondeo debemos introducir y así asegurar el correcto funcionamiento del programa. Siendo esta positiva en la dirección de la normal de los planos y negativa en la dirección contraria.

Al tener los objetos unas geometrías asociadas, podemos sacar de manera sencilla su envolvente esférica, la cual usaremos para la detección a nivel grueso, siendo esta simplemente la comprobación de que la suma entre los centros es mayor que la suma de los radios. Si se detecta que colisionan las esferas se pasa a una detección más fina a nivel de vértice y arista, la cual devuelve unos puntos de contacto que el resolutor tendrá que tratar de manera adecuada.

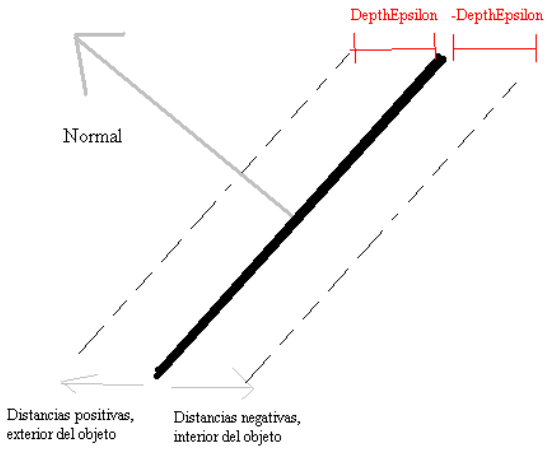


Figura 6.1: *Explicación de DepthEpsilon*

6.1.1. Algoritmo de comprobación de colisiones.

Tras haber examinado todos los objetos entre sí, así como con los muros, se devolverá un estado de colisión que puede ser nada, colisionando o penetrando.

```
ESTADO CheckForCollisions( )
{
listaContactos = nueva Lista();
//Comprobamos los cuerpos conlos muros
Para cada cuerpo C hacer
      Si C no dormido entonces
            Comprobar C con parades
      sino{
            mantener C dormido
      }
}
//Comprobamos los cuerpos entre sí
Para cada cuerpo A hacer
{
      G1 geometria = A.geometria();
      Para cada cuerpo B hacer
      {
            G2 geometria = C2.geometria();
            //Aproximación a nivel grueso
            Distancia= distacia centro A-centro B
            si (distancia()< G1.Radio()+G2.Radio()) entonces
            {
                  //Aproximación a nivel fino
                  Comprobar vértices de A y planos de B;
                  Comprobar aristas de A y planos de B;
            }
      }
   }
return ESTADO;
}
```

6.1.2. Puntos de contacto.

Antes de empezar a definir las colisiones definimos la estructura asociada a los puntos de contacto, con la cual calcularemos la respuesta a las colisiones:

- Las referencias a los **objetos** implicados. Estos pueden ser 2 cuerpos (A y B) o un cuerpo y un muro, en cuyo caso la referencia del objeto B será NULL.
- El **punto de contacto.**
- La **normal**, por representación arbitraria hemos elegido que siempre sea de B hacia A, ya que como se verá más adelante comprobamos los planos de la geometría de B con puntos pertenecientes a la geometría de A.

6.1.3. Colisión Objetos – Muros.

Es la primera comprobación y la más sencilla, con ella nos aseguramos que los objetos se encuentran dentro de los límites del mundo. Estos muros consisten en 4 planos infinitos que delimitan un espacio en el cual, los objetos que creamos interactúan, sin embargo es una representación arbitraria y se podría crear cualquier otra representación de manera sencilla.

Para la detección a nivel grueso, vamos recorriendo los muros y tras sacar su posición respecto al centro de masas de los objetos, calculamos si esta distancia es menor a la esfera asociada, en tal caso entramos a una detección más fina.

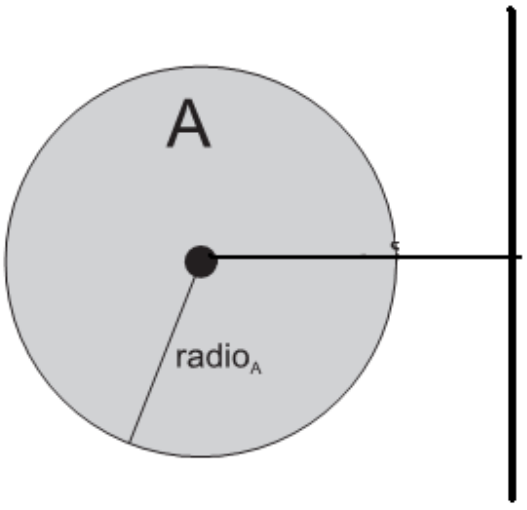


Figura 6.2: *Envolvente esférica de A y su comprobación con un muro*

Para la detección a nivel fino vamos a ir recorriendo los vértices del cuerpo, y comprobando si estos están penetrando o colisionando, para ello sacamos la posición del vértice y calculamos su distancia al muro. Si la distancia obtenida es menor que -DepthEpsilon implica que el vértice está penetrando, en el que caso que sea menor que DepthEpsilon implica que está colisionando por lo que se creará un punto de contacto que se introduce en la lista y se pasará al resolutor. El uso de DepthEpsilon se debe a como explicamos antes, a los errores de redondeo.

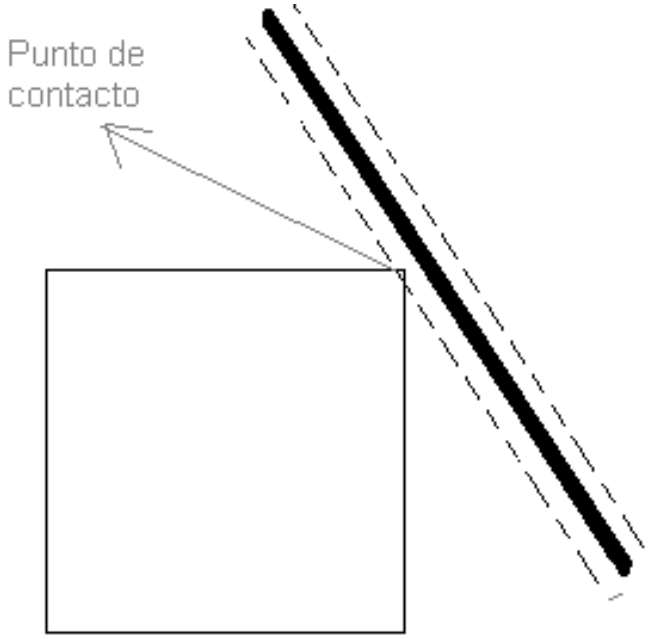


Figura 6.3: *Punto de contacto en intervalo DepthEpsilon*

6.1.4. Colisión entre objetos.

Para la colisión entre objetos, se pueden diferenciar dos fases. En una primera fase, diferente para cada tipo de colisión, se calculan una serie de puntos candidatos del objeto A que podrían ocasionar colisión con algún plano del objeto del B, estos puntos son introducidos en una lista. En la segunda parte, común a todos los tipos, se analizarán esos puntos y se decidirá si el objeto penetra, colisiona o se debe seguir con la simulación.

6.1.4.1 Obtención puntos candidatos Vértice-Plano.

Los puntos candidatos en este caso, no serán otros que los vértices del objeto A

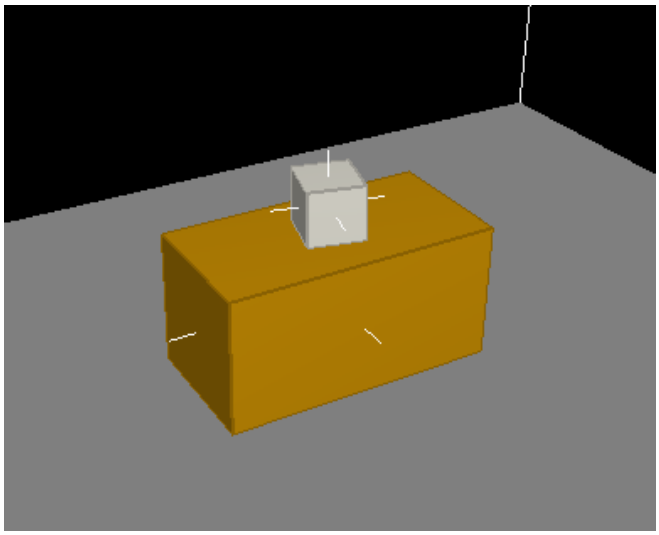


Figura 6.4: *Ejemplo contacto vértice-plano*

6.1.4.2. Obtención puntos candidatos Arista-Plano.

En este caso, la obtención de los puntos será más complicada, ya que tendremos que situar los puntos en mitad de las aristas del objeto A. Los puntos serán la intersección de las rectas infinitas de las aristas del objeto A, con los planos infinitos de las caras del objeto B. Para ello:

- Comprobamos que las rectas que contienen a las aristas de A, se cortan con los planos que contienen a las caras de B. Para ello obtenemos las ecuaciones de la recta mediante un punto **p** (en este caso un vértice) y el vector director **u,** y las del plano mediante su normal **n** y la distancia **d.**
- Calculamos el valor **t**:

$$t = \frac{-d \ - | \, n \, x \, p |}{| \, n \, x \, u \, |}$$

Si **t** es distinta de infinito, indica que se cortan, por lo que calculamos dicho punto mediante su sustitución en la ecuación de la recta.

- Ahora comprobamos que el punto se encuentra entre los 2 vértices que delimitan la arista, y en caso afirmativo introducimos el punto calculado en la lista de candidatos

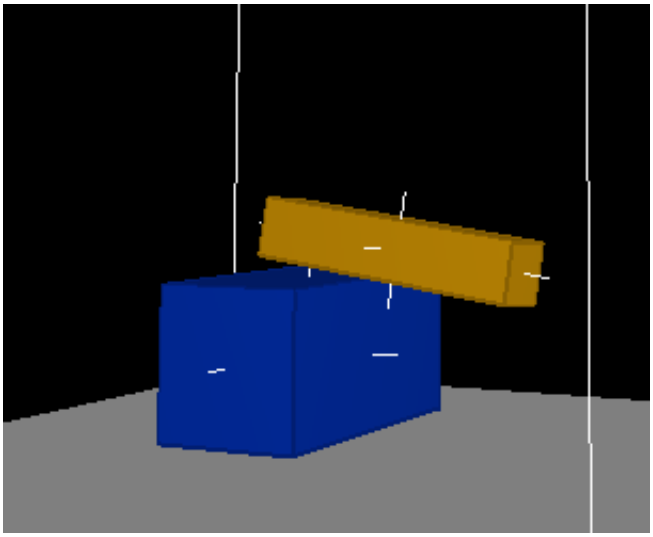


Figura 6.5: *Ejemplo contacto arista-plano*

6.1.4.3 .Clasificación de los puntos candidatos.

Para decidir cuál es el estado de los puntos candidatos, haremos un algoritmo muy parecido al de la comprobación con los muros, pero en este caso al tener varios planos tendremos que decidir cuál es exactamente con el que colisiona el punto. Para ello, con cada punto candidato haremos:

- Vamos recorriendo las caras del objeto B, si su distancia al punto es mayor que DepthEpsilon, directamente pasamos a otra cara ya que indica que punto está la suficientemente alejado.

- En caso de que sea menor, guardamos su distancia y comprobamos si su distancia es mayor que la distancia al resto de caras hasta ese momento examinadas, esto se debe a que las normales de los planos de las caras apuntan hacia fuera del objeto, luego en su interior las distancias respecto a los planos serán negativas.

- Una vez que nos hemos quedado con la cara que dista a mayor distancia, comprobamos si es menor que –DepthEpsilon y si el objeto A se encuentra entrando o saliendo del objeto B, el primer caso se tomará como penetración,

sin embargo en el segundo se descartaría el punto y se seguiría con la simulación ya que significa que se ha dado respuesta a esa penetración.

- En el caso que la distancia quede dentro del umbral de tolerancia (entre 0 y DepthEpsilon), se vuelve a comprobar si los objetos se alejan o se acercan. Si se alejan se descarta el punto y se continúa con la simulación, en el caso que se acerquen se crea el punto de contacto que se pasará al resolutor.

- Para saber si los objetos se acercan o se alejan, se comprueba la velocidad relativa del choque, tal como explicamos en las nociones teóricas.

6.2. Respuesta a las colisiones.

Lo primero, es pedir al motor de colisiones el que estado se encuentra. Si el motor está en estado de penetración, se divide la DeltaTime a la mitad, y se vuelve a comprobar el estado del sistema. De este modo se intenta llegar al instante en el que objeto que causaba la penetración todavía no haya llegado hasta tal punto y se encuentre en estado de colisión.

Si el motor está en estado de colisión, se llama al resolutor que con las lista de contactos realizara las acciones adecuadas.

Otro de los puntos a destacar en la implementación, es que tras haber resuelto las colisiones y comprobar que los objetos están estables, pasan a un estado "dormido", dando con esto estabilidad al sistema.

6.2.1. Resolutor de colisiones.

Tras comprobar que no nos encontramos en penetración, vamos recorriendo la lista de contactos, para dar respuesta a los mismos.

Lo primero, es comprobar la velocidad relativa de los objetos, en el caso que este dentro del intervalo $-\zeta<0<\zeta$, con $\zeta<<<0.01$, significa que los objetos están estables y por lo tanto se duermen, es decir, su velocidad tanto linear como angular las

forzamos a 0, con esto dotamos al sistema de mayor estabilidad. Los objetos permanecerán dormidos hasta que una perturbación exterior (un choque o la aplicación de una fuerza externa) los despierte.

En caso contrario, pasamos a calcular el impulso del choque, que como se explico anteriormente, se calcula:

$$J = j\boldsymbol{n}(t_0)$$

Acto seguido, y siempre que los objetos no sean inmóviles, modificamos las velocidades lineales y angulares de los objetos, sustituyendo la J en las fórmulas que varían tanto la velocidad lineal como la angular, actualizando las mismas y continuamos con la simulación. El impulso será negativo para el objeto B y positivo para el objeto A, debido a como ya explicamos en la parte teórica, se consideran los planos de B y por tanto su normal. Si el objeto B es un muro, se calcula el impulso sin tener en cuenta las partes de la fórmula que impliquen a dicho objeto y se actualiza solo el objeto A.

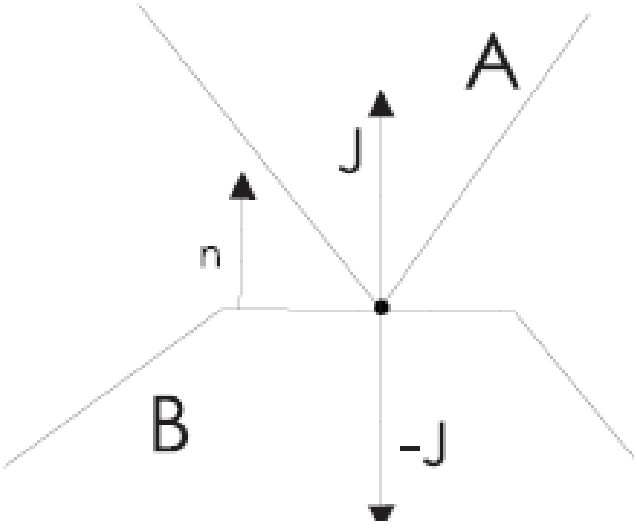


Figura 6.6: *Impulso*

7. Ejemplificación del bucle de simulación.

Para entender el funcionamiento del motor, vamos a explicar cuál es el recorrido de un paso de simulación de manera detallada.

Antes de empezar simular los objetos, el resolutor divide el tiempo a simular en intervalos del tamaño de paso especificado y los computa consecutivamente. De este modo reducimos el error resultado del integrador utilizado, de modo que si el tiempo a simular fuese 8 segundos y nuestro tamaño de paso de simulación máximo fuese 2s, en el caso de utilizar el integrador de Euler para el que el error está estimado en $error(euler) = deltaTime^2$ tendríamos:

$$error(8s) = 8^2 = 64$$

$$error(2s) + \text{error}(2\text{s}) + \text{error}(2\text{s}) + \text{error}(2\text{s}) = 2^2 + 2^2 + 2^2 + 2^2 = 16$$

De esta manera se consigue disminuir el error de manera considerable.

```
void Simular (Tiempo )
{
      real DeltaTime;
      mientras (LastTime < Time)
      {
        DeltaTime = Time - LastTime;

        Si (DeltaTime > MaxTimeStep) entonces
        {
            DeltaTime = MaxTimeStep;
        }

      //Simulamos todas las fuezas que actuan en el mundo y
      guardamos la suma de todas.

      para todas las fuerzas hacer
      {
            Simular paso de fuerzas;
            Añadir al acumulador
      }
      SimularCuerpos(DeltaTime);
      SimularParticulas(DeltaTime);
      LastTime += DeltaTime;
    }
}
```

Una vez dividido el tiempo de simulación en intervalos, para cada intervalo se simula:

```
void SimularCuerpos(real DeltaTime)
{
      real CurrentTime = 0.0;
      real TargetTime = DeltaTime;

      mientras(CurrentTime < DeltaTime) hacer{

            //Aplicamos fuerzas acumuladas a los cuerpos
            computarFuerzas();

            //Integramos los cuerpos para sacar sus nuevas posiciones
            integrarCuerpos(TargetTime-CurrentTime);

            //Calculamos la posición de los vértices
            calcularVertices();

            //Comprobamos las colsiones con las nuevas posiciones
            comprobarColisiones();

            si (ESTADO == Penetrando) entonces
            {
                  //Dividimos el tiempo entre dos
                  TargetTime = (CurrentTime + TargetTime) /2;
             }
            sino
            {

                  si(ESTADO == Colision) entonces
                  {
                      //Resolvemos las colisiones existentes
                      Motor de colisones->ResolverColision

                      //Actualizamos el tiempo
                      CurrentTime = TargetTime;
                      TargetTime = DeltaTime;
                   }
    }
}
```

Por cada intervalo se va a intentar realizar un paso de simulación del tiempo total del intervalo, si el motor de colisiones detecta algún problema (penetración) entonces desharán los cambios y se dividirá el tiempo del intervalo en dos para intentar encontrar el instante anterior al problema y resolverlo.

8. Integración del motor en una aplicación.

Para crear una prueba o aplicación que utilice nuestro motor es necesario añadir los ficheros con el código fuente del motor al proyecto o prueba para posteriormente poder incluirlos.

Seguidamente, solo hay que incluir en las clases que lo vayan a utilizar el fichero Physics.h mediante la directiva 'include': #include "Physics/Physics.h". De este modo ya se podrán utilizar todas las funciones públicas que ofrece el motor y que se explicarán posteriormente.

Para simplificar su uso ofrecemos una clase que permite crear pruebas de un modo sencillo bajo OpenGL, encapsulando la gestión de teclado, la inicialización de OpenGL, la cámara y tratando los eventos más importantes.

Esta clase que llamamos "OpenGL" sirve crear diferentes pruebas, solo hay que crear ficheros que hereden de OpenGL y sobrescribir los métodos que se desee.

Los métodos que implementa, los cuales pueden sobrescribirse para cada prueba son los siguientes:

- Inicialización de la escena, aquí se deben crear el mundo que se va a simular, establecer las características de la simulación y añadir los objetos que se van a utilizar.

```
virtual bool Init();
```

- Destrucción de la escena, creado básicamente para liberar memoria, aquí se deben destruir todos los objetos.

```
virtual void Uninit();
```

- Gestión de eventos, en éste método deben gestionarse los eventos que se desee, como el manejo de teclado, cambios en el mundo o la lógica del programa.

```
virtual void Idle();
```

- En éste método se indica que se dibuja y cómo se dibuja.

```
virtual void GlScene();
```

- Por último el método de simulación: Aquí se debe llamar al simulador utilizado, tiene un parámetro que indica el tiempo que ha pasado desde el último paso.

```
virtual void Run(float time);
```

El código que permite instanciar una prueba es el siguiente, suponiendo que hemos creado la prueba "Test1" que hereda de "OpenGL" en el main se escribe lo siguiente:

```
OpenGL * opengl = ((Test1*)opengl)->create();
if (!opengl->initGL("OpenGL Window Test1", 800, 600, false, 32))
      return 1;
opengl->runMessageLoop();
```

Con estas líneas hemos creado una ventana OpenGL de 800x600x32 de título "OpenGL Windows Test1" utilizada para simular la prueba "Test1". Ya solo hay que rellenar los métodos heredados.

9. Conclusiones.

A la hora de realizar éste proyecto, debido a su complejidad, tuvimos que plantear una serie de etapas claramente diferenciadas, que nos ayudaran a llevar una trayectoria correcta para crear una API que permitiese la simulación de la dinámica de partículas y la dinámica de sólidos con colisiones.

En primer lugar se realizó un estudio completo y documentado del estado actual de los motores de física más conocidos así como de las nociones de análisis numérico, física y geometría que aplicaríamos en fases posteriores. Paralelamente se creó una sencilla aplicación OpenGL con que nos permitiese visualizar el estado del mundo que íbamos a simular para utilizarlo posteriormente en las pruebas de simulación.

Para el siguiente paso decidimos comenzar con el primer módulo de nuestro motor que integraría únicamente la dinámica de partículas. Ésta primera versión permite crear un "mundo" vacío al que se pueden añadir fuerzas y sistemas de partículas y simular su comportamiento. Se crearon dos ejemplos de sistemas de partículas: explosiones y corrientes de partículas.

Finalmente realizamos una batería de pruebas de estabilidad y rendimiento. En las pruebas de estabilidad los resultados fueron muy buenos, el sistema era muy estable debido a la sencillez de la dinámica de partículas. En cambio las pruebas de rendimientos presentaban problemas para sistemas sobrecargados de partículas en los que se observaba que el sistema se ralentizaba.

Éste problema podía solucionarse aumentando el tiempo de paso de la simulación, pero esté aumento va ligado a una reducción del realismo. Como no podemos saber de antemano que necesidades va a tener el usuario final del motor, decidimos parametrizar éste valor y así cumplir con el compromiso entre realismo y tiempo de cálculo.

Sobre ésta primera versión ya podíamos trabajar en la siguiente, extendiendo el comportamiento de las partículas y creando un módulo para la simulación de sólidos

rígidos. Añadiendo los conceptos de rotación y geometría creamos fácilmente la siguiente versión que encapsularía la dinámica de partículas y sólidos.

Una vez finalizada ésta versión volvimos a testear su estabilidad y rendimiento obteniendo resultados satisfactorios en estabilidad pero surgieron problemas de rendimiento. En sistemas con muchas partículas y sólidos había conflicto con el paso de simulación, si éste era grande el rendimiento era bueno pero se perdía mucho realismo en la simulación de sólidos, si era pequeño se perdía mucho tiempo en la simulación de las partículas.

Las partículas al contrario que los sólidos no necesitan un tamaño de paso pequeño para tener un comportamiento realista, además no sabemos de antemano el tipo de aplicación que utilizará nuestro motor ni sus necesidades por lo que la mejor solución a éste problema fue separar los pasos de simulación de sólidos y partículas y parametrizarlos para que sea el usuario final el que decida su valor.

En una siguiente versión añadimos los conceptos que nos permitieron añadir las uniones elásticas, tras realizar las pruebas se obtuvieron buenos resultados de rendimiento y volvimos a tener algunos problemas de estabilidad. En ocasiones, debido al error acumulativo del integrador de Euler el sistema "explotaba", dado que éste problema solo ocurría para determinadas pruebas no se solucionó, y se deja para un futuro implementar integradores más avanzados.

El siguiente módulo que añadimos fue el módulo para la detección y respuesta de colisiones, comenzando con las colisiones más sencillas que no presentaban un comportamiento demasiado realista para finalizar con las más complejas y precisas. Tras las pruebas del módulo se observaron buenos resultados de rendimiento y estabilidad para la mayoría de los casos, en los casos más problemáticos el sistema no es capaz de resolver la interpenetración, pero como vimos en el estudio previo incluso los mejores motores profesionales tienen este tipo de fallos, que solucionan mediante técnicas que los ocultan.

El último módulo que decidimos añadir fue un módulo adicional que permitiese trabajar con uniones entre partículas y crear sistemas de partículas complejos, y así dar soporte a futuras ampliaciones como la simulación de telas y sólidos deformables. Para probar que el módulo funcionaba, intentamos simular una tela obteniendo muy buenos resultados en dicho test.

Una vez finalizado y testeado el motor nos vimos ante el problema de que era muy difícil crear una aplicación desde cero que lo utilizase, por lo que decidimos implementar un proyecto aparte, que mediante un modelo vista-controlador encapsula la inicialización de OpenGL, el tratamiento de las interrupciones de teclado y el bucle típico de aplicaciones de simulación , haciendo éste uno de los puntos fuertes de nuestro proyecto debido a la sencillez con la que se pueden crear aplicaciones gráficas con nuestro motor integrado.

Como conclusión final, hemos realizado el esqueleto de un motor multifuncional que puede ser ampliable de manera sencilla, habiendo obtenido buenos resultados de rendimiento y estabilidad, además de un comportamiento realista, también se ha llegado un nivel bastante profundo en varios aspectos como en el tratamiento de las colisiones, llegando a un nivel fino o el tratamiento de partículas que pueden unirse para formar objetos complejos. Además todos los módulos están pensados para facilitar futuras ampliaciones y la realización de proyectos que integren nuestro motor es realmente sencilla como ya explicamos antes.

9.1. Futuras Mejoras y ampliaciones.

El desarrollo de un motor de física es un proceso largo y complejo, cuyas posibilidades de ampliación son elevadas, pero con ello también los problemas que surgen. Incluso los motores profesionales están en continuo desarrollo y no exentos de esta problemática.

A continuación se enumeran una serie de aspectos a mejorar:

- Implementar y comparar distintos tipos de integradores, como Runge-Kutta, lo que mejorará el error en los problemas de cálculo asociados a la simulación.

- Buscar salidas al motor de colisiones en los momentos que no sea posible proseguir con la simulación.

- Introducir nuevas formas como esfera, pirámides o cilindros, o incluso la posibilidad de cargar modelos de mallas convexas con software 3DMax, en vez de implementarlas sobre la clase CGeoBox.

- Introducir nuevos tipos de uniones elásticas e inelásticas que permitan crear objetos compuestos complejos o articulados al combinarse con los sólidos y partículas.
- Pueden crearse uniones rígidas e incluso rompibles que permitirán crear nuevos materiales como fluidos, telas, cristal y efectos de rotura de los mismos.

- A partir de sistemas de partículas y las nuevas uniones crear otro tipo de objetos como sólidos deformables, materiales de distintos tipos, fluidos, gases...

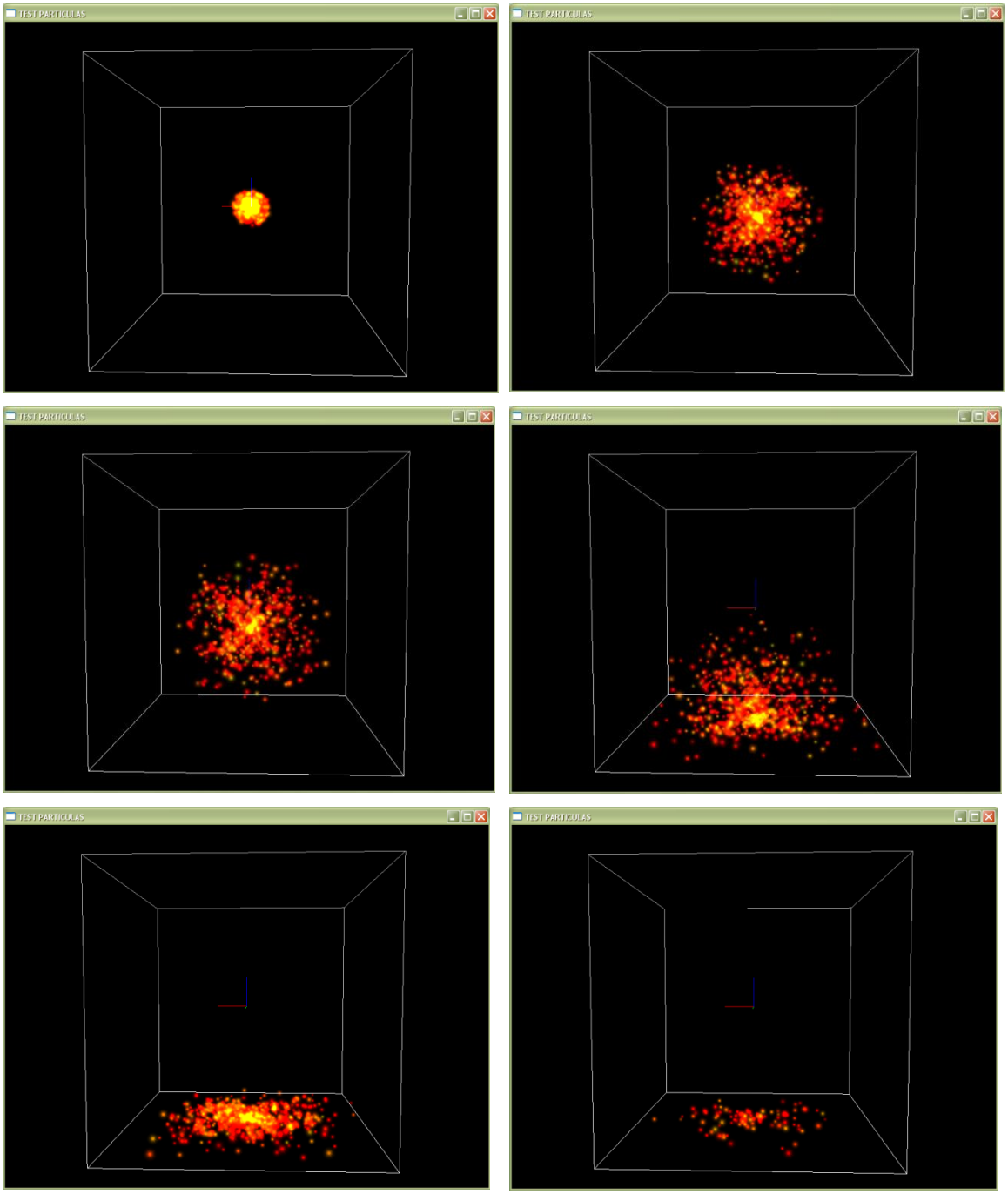
- Ampliar el módulo de colisiones para que detecte y resuelva las colisiones entre partículas y sólidos.

Otras mejoras

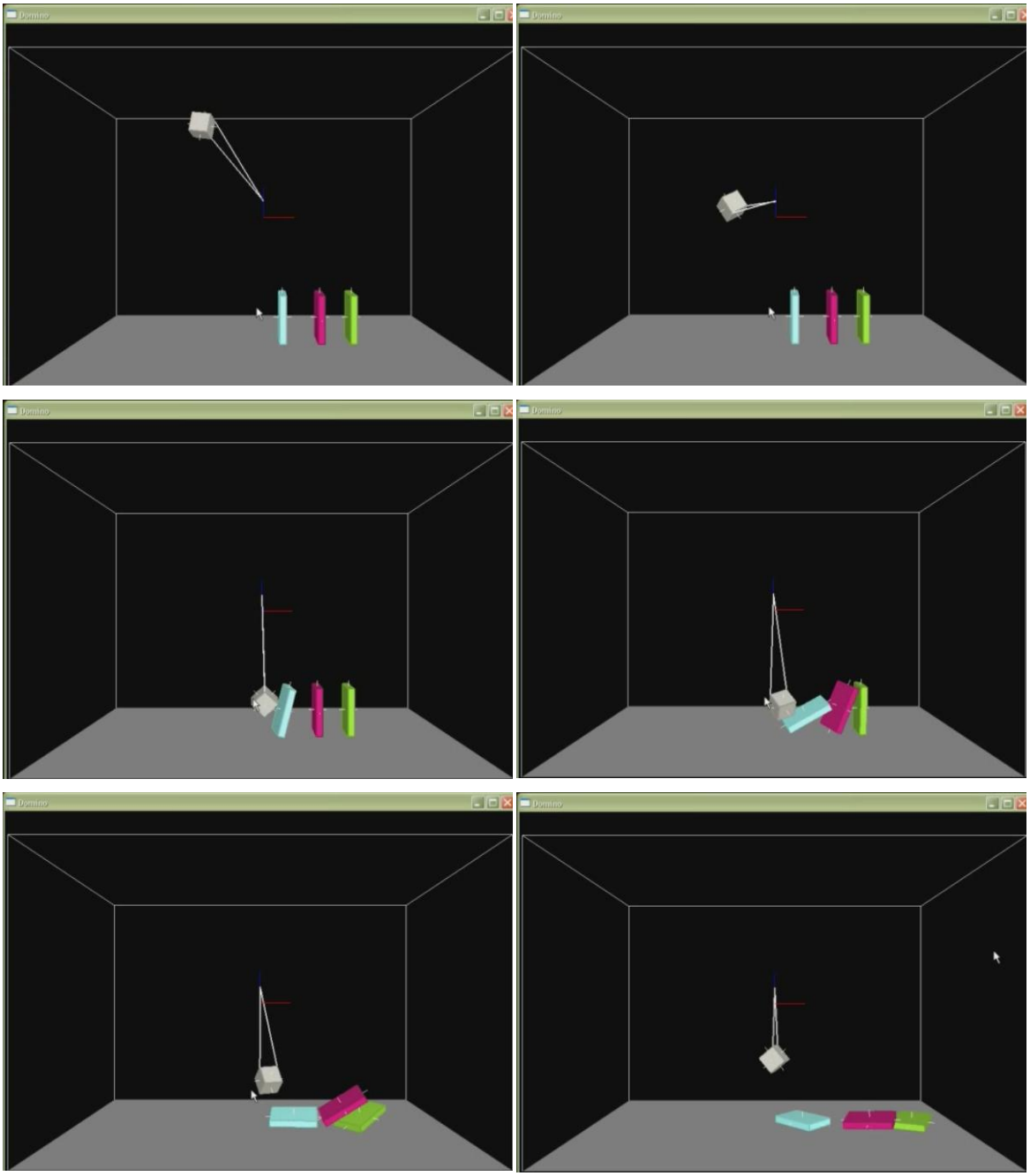
- Propagación de eventos.
- Más niveles de detección de colisiones.
- Dinámica con restricciones.

10. Ejemplos de pruebas

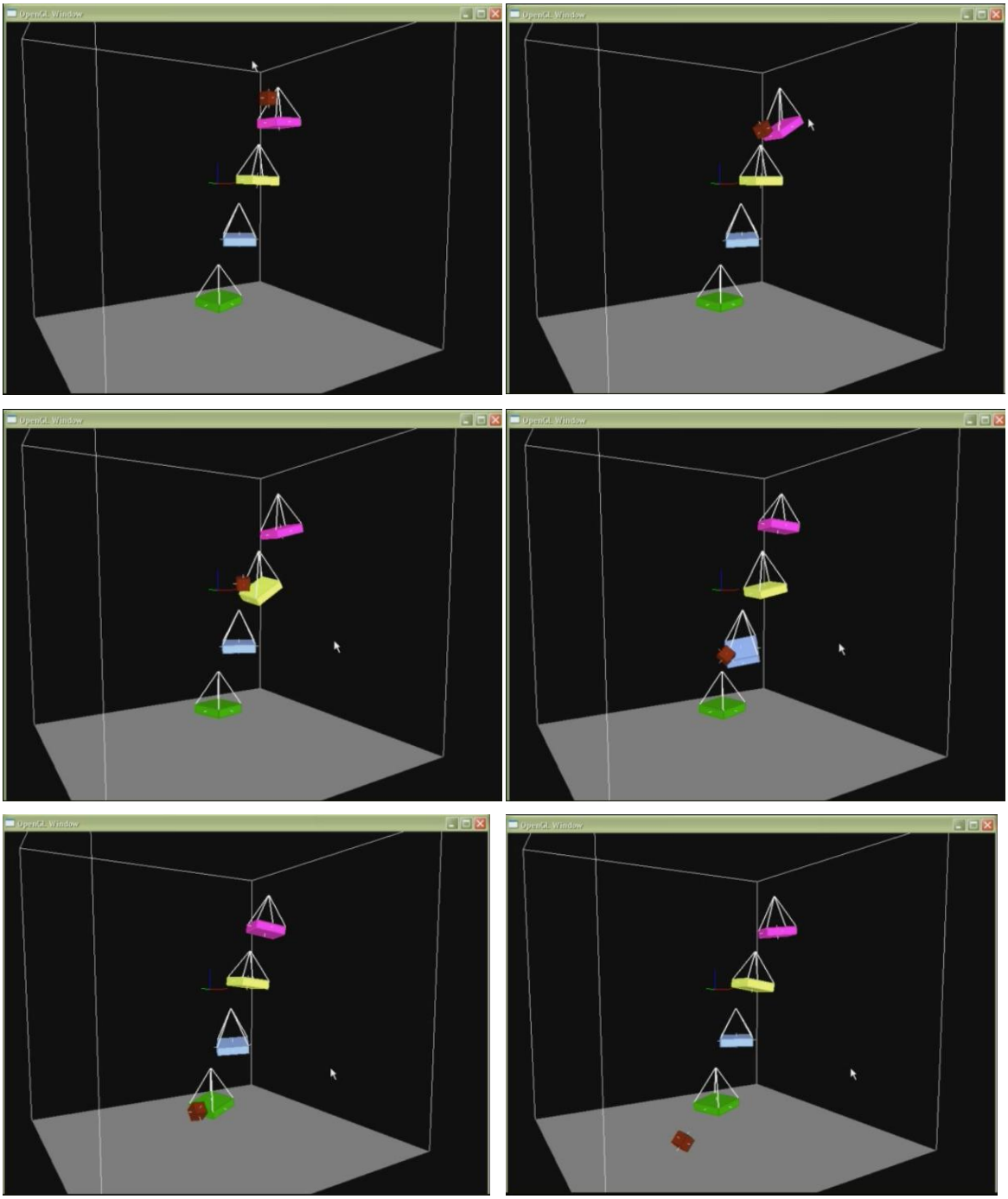
En esta prueba se puede ver el ciclo de vida de una explosión. Su nacimiento, su expansión bajo los efectos de la gravedad y el rozamiento y finalmente su desaparición.



En este ejemplo, se tiene un péndulo compuesto de un solido atado a la pared mediante Springs, que en su caída, debida a la aceleración de la gravedad golpea una ficha que genera un efecto dominó empujando a la siguiente para llegar al final a un estado de reposo. Este ejemplo es bastante ilustrativo pues contiene fuerzas, springs, sólidos y colisiones.



Para ésta prueba se han dispuesto cuatro plataformas móviles creadas con la ayuda de Springs, dejamos caer un objeto sobre la primera y vemos como éste va descendiendo por todas, alterando su estado debido a las colisiones que se generan.



11. Bibliografía.

- David Baraff. Particle Dynamics**.** En An Introduction to Physically Based Modelling. SIGGRAPH ’97 course notes., 1997.

- David Baraff. Differential equation basics. En An Introduction to Physically Based Modelling. SIGGRAPH ’97 course notes., 1997.

- David Baraff. Rigid body simulation. In An Introduction to Physically Based Modelling. SIGGRAPH ’97 course notes., 1997.

- David M. Bourg. Physics for Game Developers. O’Reilly, primera edición, Noviembre 2001.

- David H. Eberly Game Physics segunda edición. Noviembre 2006.

- David H. Eberly 3D game engine design : a practical approach to real-time computer graphics. San Francisco : Morgan Kaufmann, cop. 2007

- http://www.ode.org/

- http://developer.nvidia.com/object/physx.html

- http://www.havok.com

- http://www.cidse.itcr.ac.cr/cursos-linea/SUPERIOR/algebra-vectorial-planos-rectas/node5.html

- http://graphics.stanford.edu/courses/cs448-01-spring/papers/moore.pdf

- http://huitoto.udea.edu.co/Matematicas/4.11.html

- http://www.geoan.com/analitica/distancias/distancia_rectas.html

- Segundo Esteban. Física de Videojuegos, Apuntes del Master en Desarrollo de Videjuegos,

- Alan Watt & Fabio Policarpo "3D Games", Addison-Wesley, 2001.

- R. D. Cook, D. S. Malkus and M. E. Plesha Concepts and Applications of Finite Element Analysis, R. D. Cook, D. S. Malkus and M. E. Plesha, John Wiley & Sons, 1989.

- K.-J. Bathe Finite Element Procedures , Prentice Hall, 1996.

- Y.C. Fung ,First Course in Continuum Mechanics , Prentice Hall, 1993.

- Press, Flanner, Teukolsky and Vetterling. Numerical Recipes, Cambridge University Press.

- Ciarlet and Lions. Handbook of Numerical Analysis , Vol. I - VI, North-Holland, 1994.

Los abajo firmantes autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Jaime Sánchez-Valiente Blázquez Santiago Riera Almagro Matías Salinero Delgado